VIRTUALIZATION SERVICES: SCALABLE METHODS FOR VIRTUALIZING MULTICORE SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Himanshu Raj

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
April 2008

VIRTUALIZATION SERVICES: SCALABLE METHODS FOR VIRTUALIZING MULTICORE SYSTEMS

Approved by:

Prof. Karsten Schwan, Adviser
College of Computing
Georgia Institute of Technology

Prof. Henry Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Ada Gavrilovska
College of Computing
Georgia Institute of Technology

Prof. Richard Fujimoto
College of Computing
Georgia Institute of Technology

Jimi Xenidis
Advanced Operating Systems Research Department
IBM T.J. Watson Research Labs, Yorktown Heights, NY

Date Approved: 7 January 2008

To my parents, whose faith in me is and will always be the source of my strength.

ACKNOWLEDGEMENTS

I am beholden to my advisor, Prof. Karsten Schwan, for his infallible guidance throughout my PhD tenure. His cheerful recognitions, endless support, and continuous encouragement has immensely helped me in my research. I would also like to thank other committee members for help and feedback. Jimi Xenidis provided me with opportunities to work with him early on in the area of virtualization, which formed the basis for much of my research. Dr. Ada Gavrilovska has been a great collaborator and mentor for the work related to network processors.

I acknowledge help and input from other students in the group, most notably from Balasubramaniam Seshasayee, Sanjay Kumar, Ivan Ganev, Ripal Nathuji, and Sandip Agarwala. I also greatly appreciate the joyous banter that we had at times, without which it will be all work and no play.

I am thankful to Jeffrey Daly at Intel and Kenneth McKenzie at Reservoir Labs for their help with implementation issues regarding the 21555 bridge and the network processor platform. Parts of this work were motivated by research interactions with Jun Nakajima and Rob Knauerhase at Intel Corp., and Hubertus Franke at IBM T.J. Watson Research Labs.

Last, but certainly not the least, I acknowledge the constant love and support from my family, in particular from my wife, Richa. Her encouragements and savory food help keep me sane in otherwise overwhelming times. Without her by my side, surviving a PhD at Tech would have been difficult.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Multi-core technology is bringing parallel processing capabilities from servers to laptops and even handheld devices. At the same time, platform support for system virtualization is making it easier to consolidate server and client resources, when and as needed by applications. This consolidation is achieved by dynamically mapping the *virtual machines* on which applications run to underlying physical machines and their processing cores. Low cost processor and I/O virtualization methods efficiently scaled to different numbers of processing cores and I/O devices are key enablers of such consolidation.

This dissertation develops and evaluates new methods for scaling virtualization functionality to multi-core and future many-core systems. Specifically, it re-architects virtualization functionality to improve scalability and better exploit multi-core system resources. Results from this work include a *self-virtualized I/O* abstraction, which virtualizes I/O so as to flexibly use different platforms' processing and I/O resources. Flexibility affords improved performance and resource usage and most importantly, better scalability than that offered by current I/O virtualization solutions. Further, by describing system virtualization as a service provided to virtual machines and the underlying computing platform, this service can be enhanced to provide new and innovative functionality. For example, a virtual device may provide obfuscated data to guest operating systems to maintain data privacy; it could mask differences in device APIs or properties to deal with heterogeneous underlying resources; or it could control access to data based on the "trust" properties of the guest VM.

This thesis demonstrates that extended virtualization services are superior to existing operating system or user-level implementations of such functionality, for multiple reasons. First, this solution technique makes more efficient use of key performance-limiting resource in multi-core systems, which are memory and I/O bandwidth. Second, this solution technique better exploits the parallelism inherent in multi-core architectures and exhibits good

scalability properties, in part because at the hypervisor level, there is greater control in precisely which and how resources are used to realize extended virtualization services. Improved control over resource usage makes it possible to provide value-added functionalities for both guest VMs and the platform. Specific instances of virtualization services described in this thesis are the network virtualization service that exploits heterogeneous processing cores, a storage virtualization service that provides location transparent access to block devices by extending the functionality provided by network virtualization service, a multimedia virtualization service that allows efficient media device sharing based on semantic information, and an object-based storage service with enhanced access control.

CHAPTER I

INTRODUCTION

1.1 Motivation

The multicore nature of computer architectures is extending the multiprocessing capabilities of server systems to lower end desktops, laptops, and even handheld devices. At the same time, the increased use of virtualization is making it possible to map to these platforms, efficiently and simultaneously, the computing workloads of multiple and often highly diverse applications. It is well-known that this flexibility enables the efficient use of server resources, through runtime server or service consolidation, thereby reducing overall cost of hardware ownership by reducing the number of servers required during peak usage periods. This thesis extends such knowledge to demonstrate that in addition, the combined capabilities of multi-core machines and virtualization technology can provide end users with new and high performance services, by seamlessly exploiting the combined resources of both the end systems being used and the server infrastructures that support these services.

Server consolidation and the seamless provision of services to end users rely on several key properties of virtualization. These include the *performance isolation* necessary to permit two applications to share the same physical resource in a predictable manner, and the reliability and trust guarantees for the *fault isolation* needed to prevent failures or misbehavior in one application from affecting others. Guarantees are attained by embedding applications in different, isolated *Virtual Containers*, with isolation guarantees provided by the underlying *Virtualized Platform* (VP), either in software and/or hardware. Each container has its separate virtual architectural resources, including both core resources such as virtual CPUs and memory, and peripherals, such as virtual disks. These virtual resources are mapped to the architectural resources of one or more physical platforms via a variety of methods, including resource partitioning, time sharing, or a combination thereof. These

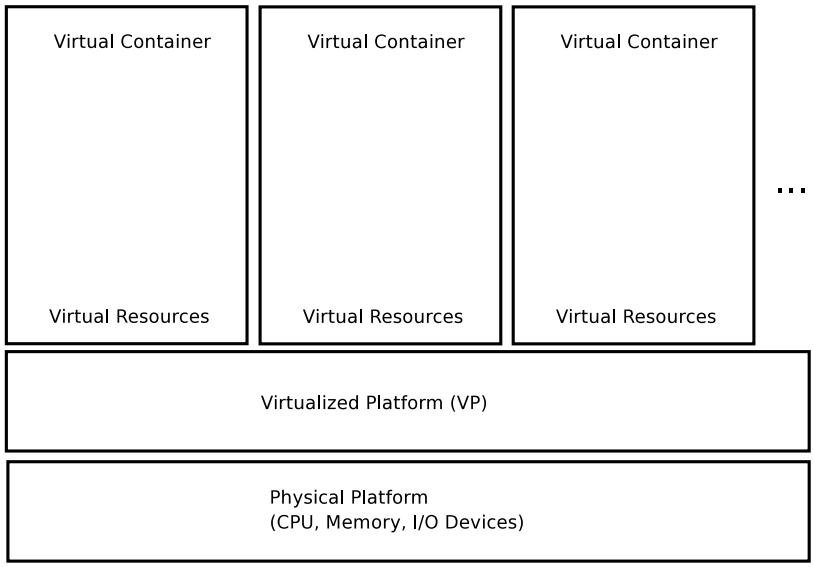


Figure 1: Overview of system virtualization.

methods are used to create virtual instances of all physical resources and dynamically manage these virtualized components among the multiple virtual containers. Figure 1 depicts a high level view of system virtualization.

Virtual containers differ with respect to the operating environments they provide to applications and in the mechanisms utilized to impose isolation, both of which impact the performance and scalability of each specific virtualization solution. For example, with *hypervisor-level virtualization* technology, these containers, called *Virtual Machines* (VMs) or *domains*, execute on a VP managed by a *hypervisor* (HV) or a *virtual machine monitor* (VMM), and possibly one or more privileged Service VMs. In contrast, OS-level virtualization provides *Virtual Environments* (VEs) or *Virtual Private Servers* (VPSs), on a VP managed by an OS kernel. The operating environments provided by VMs provide more flexibility, since each VM can run a different OS kernel, albeit at a higher impact on performance and scalability [104]. In contrast, VEs must share the same underlying host OS kernel. Examples of the former technology include VMWare [36] and Xen [48], examples of the latter include OpenVZ [25] and Linux VServers [20]. Hypervisor-level virtualization can obtain significant benefits from recent and upcoming architectural advances in hardware support for virtualization (*e.g.* Intel's VT [17] and AMD's Pacifica [4] technologies)

and I/O virtualization support from upcoming PCI devices [27].

Hypervisor-level virtualization technology can be further sub-divided into full vs. para-virtualization. In the case of full virtualization, an unmodified OS kernel designed to execute on bare hardware can be used inside a VM [36]. In contrast, para-virtualization requires VMM-specific modifications to the OS kernel [48]. Para-virtualization can provide substantial performance benefits over full virtualization since it can avoid costly runtime emulation of privileged instructions, albeit it requires extensive modifications to the OS kernel. Limited form of para-virtualization, specifically for I/O, can enhance the performance for legacy OSes [126]. Hardware assisted virtualization can further provide performance benefits for fully and partially para-virtualized guests, without requiring extensive and costly modifications to the guest VM's OS kernel [44]. Another way to categorize the hypervisor-level technology is Type 1 HV vs. Type 2 HV [12] – a type 1 HV runs directly on the physical hardware, while a type 2 HV runs on a host OS.

The focus of this work is exclusively on Type 1 hypervisor-level virtualization. Furthermore, most of the methods proposed in this work exploit para-virtualization, unless explicitly mentioned otherwise.

A challenge for all methods for system virtualization is how to deal with the multi- and many-core nature of future execution platforms, in particular the heterogeneity present in such systems. Such heterogeneity has multiple sources, including performance differences among multiple communication paths, e.g., inter-core communication channels, memory buses and I/O buses, and differences across computational cores, in terms of their speeds or capabilities or even their instruction set architectures (ISAs) [33]. Failing to deal with such heterogeneity can have substantial performance implications, such as a high number of cache misses, increased instruction counts, and the necessity to schedule multiple domains for simple system actions, which can cause high virtualization overheads [98]. In this context, a key issue identified and explored in this thesis is whether or not or better, to which degree virtualization methods should provide virtual platforms to guest VMs that closely or exactly resemble the underlying physical platform, as done by current systems. For example, currently, virtual devices tend to provide the same low-level APIs as the

physical device being virtualized. This thesis finds issue with this approach, because it forces a VM to implement any and all useful services utilizing these virtual devices, resulting in a potential mismatch between computational resources available to this VM and the computational demands imposed by these services. Another problem with a close match of virtual to physical device is that this match may remove some of the potential benefits of system virtualization, specifically efficient consolidation and device sharing. A specific example demonstrated in this thesis concerns multimedia devices, where the costs associated with current methods for time-sharing such devices may be too prohibitive to allow any meaningful sharing. Finally, the low level interfaces offered by most devices also make it difficult to provide value-added services for the platform, such as security related services that monitor a VM's behavior for any malicious activity [105].

In order to address these challenges and problems, this dissertation adopts an alternative view of a system virtualization solution, motivated by the paradigm of Service Oriented Architecture (SOA). In particular, we consider each of the different resources being virtualized as a specific service, resulting in a collection of *Virtualization Services*. A virtualization service provides a virtual architectural resource, such as CPU, memory or I/O device, along with the API needed to access this resource, to the service consumer, which resides in some Virtual Container (VC). In addition, each such service can be extended beyond simply virtualizing a resource to presenting an *enhanced* virtual resource, with enhancements concerning improved performance or platform scalability and/or additional value-added attributes and functionalities useful to VMs or required by underlying platforms. Collectively, then, these services constitute an *enhanced* virtual platform for a VM, provided at no or little additional cost to VMs. For example, a useful storage virtualization service is one that provides a virtual disk with additional reliability properties for the data stored on it, where typically, costs are commensurate with the degree of reliability offered. Hence, following the definition of a service in SOA [124], a virtualization service is a basic building block in *virtualized systems* that combines information and behavior, hides the implementation of the service provider from the consumer, and provides a well defined API as a basis for service utilization.

Service providers need not reside on the same physical platform on which the service is used, nor are they confined to single platforms. As a result, components of the service provider may or may not be a part of the specific VP hosting the service consumer. In fact, the service provider may not execute in a virtualized environment at all. A specific example is the storage service where virtual block devices being used by VCs are accessed over a network using iSCSI, and the service provider executes on a separate, non-virtualized, physical machine with attached physical storage devices. Here, the role of the VP is that of a facilitator for service consumers. On the other hand, some virtualization service providers may be an integral part of the VP hosting the consumers, such as the CPU virtualization service providing virtual CPUs to the VCs.

Before proceeding further, we digress briefly to state that this thesis defines scalability in the context of virtualization services with respect to the cost behavior of a virtualization service instantiation with increasing number of guest VMs on a fixed physical platform. Thus, a single virtualization service instance is considered scalable if the cost of service use increases linearly or sub-linearly with an increasing number of guest VMs. A meaningful way to assess scalability is by comparing two or more virtualization service realizations, where one is deemed more scalable than the other if the cost imposed on a guest VM by the former is less than the latter, for some specific number of guest VMs. The rationale, of course, is that it is possible to service a larger number of guest VMs with the former instead of the latter instance of a service, while still meeting some acceptable performance criterion. A similar approach to defining scalability is used in other virtualization research [48, 104].

This dissertation advocates and explores the paradigm of virtualization services in the context of future many-core, off-the-shelf hardware platforms, and addresses the challenges related to heterogeneity as described above. First, costs are reduced by departing from the monolithic structure of current virtualization software, since virtualization services make it possible to create custom, per-resource virtualization solutions, judiciously mapped to certain computational cores. Second, by enhancing services with new functionality, the approach naturally associates the computations needed to create semantically meaningful resources with the task of resource virtualization, thereby exploiting the computational

power of future many-core platforms. Moreover, by specializing such computations to the needs of specific resources, scalable and efficient virtualization solutions can be found.

While the virtualization services used in this dissertation match the many-core nature of future machines, a specific technical problem faced by all such software is to deal with the hardware-imposed limits on memory and I/O bandwidths present in many-core machines. This leads to the technical question of how to efficiently move data to/from the memory locations where it is needed by computational cores. Service-based virtualization addresses this question in multiple ways. First, we create *self-virtualized I/O services*, which means that all of the functionality required for virtualizing a certain I/O peripheral is embedded within a single service description. This compact description enables flexible and efficient mappings of service functionality to underlying hardware resources. Second, mappings are performed to best exploit platform resources, a key metric being the judicious use of I/O and memory bandwidths. Third, multi-core resources are used to enhance the value or utility of I/O services to guest operating systems and their applications. This includes applying these computational resources to remove the need for data copying or communication (i.e., reducing memory bandwidth or I/O needs) and/or to provide additional functionalities, including those that offer enhanced device semantics or security.

1.2 The Thesis

This dissertation's thesis is that:

A new virtualization approach and methods are needed to cope with the increasing mismatch of computational capabilities vs. memory and I/O bandwidths in future many-core machines. *Virtualization services* coupled with re-architecting core virtualization components to support these services can provide scalable performance for future platforms.

This dissertation focuses primarily on the I/O component of system virtualization. It addresses the following key research problems in this domain:

- The performance and scalability of current I/O virtualization methods is limited. Given the increasing divide between a system's computational resources and I/O

bandwidth, this problem will only become more exacerbated. How can we improve these methods?

- Current virtualization methods provide virtual devices that simply export the APIs of underlying physical device(s). This can cause inefficiencies in the implementation of certain *services* needed by guest VMs. For example, a guest VM requiring access to an iSCSI-based storage device may execute the iSCSI stack over the virtual network device. A more efficient method to provide this service may be to implement the iSCSI protocol 'inside' the network virtualization subsystem which then provides a SCSI block device interface to the guest VM. Using multi-core resources, how can we enhance virtualization methods in order to provide improved services to guest VMs?

1.2.1 Contributions

The main contributions of this dissertation are as follows:

- It defines a virtualization service as a fundamental construct to compose a system level virtualization solution. Focusing on I/O, we describe an abstraction for composing rich virtualization services, termed self-virtualized I/O (SV-IO). SV-IO encapsulates all of the tasks associated with virtualizing an I/O device: it provides virtual devices and associated access API to guest VMs and management API to the VMM. These virtual devices can then be *extended* with enhanced functionality.

- The SV-IO abstraction provides flexible ways to realize peripheral virtualization. In particular, it allows *device-centric* realizations by using processing capabilities that might be present *closer* to the peripheral device, while also permitting the more traditional *host-centric* realizations. Implementations of these realizations for a gigabit network interface with on-board processing resources (based on the IXP2400 network processor), i.e., for a prototype heterogeneous multicore platform, demonstrate that a device-centric realization exhibits improved performance ($\sim 2X$ better throughput, higher scalability, and $\sim 50\%$ less latency) than a host-centric realization.

- The SV-IO abstraction allows us to partition virtualization responsibilities to specific

cores. In a future multicore setting, this approach can provide significant performance benefits, as demonstrated by further experimental results. In particular, dedicating a specific host core for interrupt virtualization along with the device-centric network virtualization service provides up to 50% latency reduction for 32 guest VMs vs. the case when interrupt virtualization task is shared by all cores. This partitioning approach, termed *Sidecore*, can be used for other virtualization services, such as the one providing page table updates and the one facilitating better VM-VMM communication in VT-enabled systems [88], and access to certain *trusted* resources, where these resources may only be available to some cores.

- I/O virtualization services based on the SV-IO abstraction provide opportunities for enhanced functionalities to guest VMs and to the platform itself. These services enhance a VM's interactions with the outside world by providing *semantically enhanced virtual devices*, termed *logical devices*. Logical devices exploit semantic information to better support the end-to-end requirements of these VM communications. Further, a virtualization service using semantic information can better aggregate multiple, possibly heterogeneous, devices.

  The benefits of enhanced virtualization services are demonstrated with logical devices using three examples: (1) a low-latency, *transparent device remoting based storage service*, assisted by a device-centric SV-IO realization for a NIC, (2) an enhanced multimedia virtualization service that provides high performance and better aggregation of multiple multimedia devices, and (3) an object-based storage architecture (*O2S2*) and its realization that provides a security-enhanced and trusted storage solution. These services not only demonstrate enhanced functionality, but also do so at low or no cost to guest VMs. Further, by exploiting semantic knowledge, they can also provide better performance. For example, the multi-media virtualization service provides substantial latency reduction over a solution based time-sharing of the multimedia device, since the latter compounds the physical device access time as overhead. Similarly, the object-based storage service can provide upto $2X$ better read

performance for large data files, as compared to a traditional storage solution built using virtual block based devices.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the self-virtualized I/O abstraction as the key component to implement virtualization services for I/O. It also describes *device-* and *host-centric* realizations of a prototype network virtualization service as concrete realizations of this abstraction, along with their performance analysis. This chapter also identifies various architectural enhancements to improve and scalability of I/O virtualization services. The SV-IO abstraction can further take advantage of the the core partitioning 'sidecore' approach in a multi-core system, as described in Chapter 3. We describe the benefits of the sidecore approach for the prototype device-centric network virtualization service described in Chapter 2, along with the benefits for other virtualization services, such as for reducing VM-VMM communication latency and for page table updates for guest VMs in VT-enabled systems.

Chapter 4 describes *logical devices* as a component of enhanced virtualization services. Next, we describe example enhanced virtualization services that provide value-added functionality to the guest VMs and the platform. Chapters 5 and 6 are devoted to two such example services for multimedia virtualization and storage virtualization, respectively.

Chapter 7 compares the work introduced in this dissertation with past and ongoing related work in the areas of I/O virtualization and composing services in virtualized environments. Chapter 8 concludes the dissertation with a summary of its contributions and explores future directions.

CHAPTER II

SELF-VIRTUALIZED I/O: HIGH PERFORMANCE AND SCALABLE I/O VIRTUALIZATION SERVICES

Virtualizing I/O subsystems and peripheral devices is an integral part of system virtualization. This chapter advocates the notion of *self-virtualized I/O* (SV-IO) for building virtualization services for I/O virtualization. Specifically, it proposes a hypervisor-level abstraction that permits guest virtual machines to efficiently exploit the multi-core nature of future machines when interacting with virtualized I/O. The concrete instance of SV-IO developed and evaluated herein (1) provides virtual interfaces to an underlying physical device, the network interface, and (2) manages the way in which the device's physical resources are used by guest operating systems. This instance provide the network virtualization service to guest VMs. The performance of this instance differs markedly depending on design choices that include (a) how the SV-IO abstraction is mapped to the underlying host- *vs.* device-resident resources, (b) the manner and extent to which it interacts with the HV, and (c) its ability to flexibly leverage the multi-core nature of modern computing platforms. A *device-centric* SV-IO realization yields a *self-virtualized network device* (SV-NIC) that provides high performance network access to guest virtual machines. Specific performance results show that for high-end network hardware using an IXP2400-based board, a virtual network interface (VIF) from the device-centric SV-IO realization provides ∼77% more throughput and ∼53% less latency compared to the VIF from a *host-centric* SV-IO realization. For 8 VIFs, the aggregate throughput (latency) for device-centric version is 103% more (39% less) compared to the host-centric version. The aggregate throughput and latency of the VIFs scales with guest VMs, ultimately limited by the amount of physical computing resources available on the host platform and device, such as number of cores. The chapter also discusses architectural considerations for implementing self-virtualized devices in future multi-core systems.

2.1 Introduction

This chapter focuses on virtualization services for I/O device virtualization, by presenting the abstract notion of a *Self-Virtualized I/O* (SV-IO). SV-IO captures all of the functionality involved in virtualizing an arbitrary peripheral device. It offers virtual interfaces (VIFs) and an API with which guest domains can access these interfaces. It specifies that the actual physical device must be multiplexed and demultiplexed among multiple virtual interfaces. It states that such multiplexing must ensure performance isolation across the multiple domains that use the physical device, and/or meet QoS requirements from guest domains stated as fair share or other metrics of guaranteed performance.

The SV-IO abstraction describes the resources used to implement device virtualization. Specifically, a device virtualization solution built with SV-IO is described to consist of (1) some number of processing components (cores), (2) a communication link connecting these cores to the physical device, and (3) the physical device itself. By identifying these resources, *SV-IO can characterize, abstractly, the diverse implementation methods currently used to virtualize peripheral devices*, including those that fully exploit the multiple cores of modern computing platforms. These methods, characterized as *host-centric* methods since all virtualization functionality executes on host processing cores, include using a driver domain per device [132], using a driver domain per one set of devices [110], or running driver code as part of the HV itself [48]. The latter approach has been dismissed in order to avoid HV complexity and increased probability of HV failures caused by potentially faulty device drivers. Therefore, current systems favor the former approaches, but performance suffers from the fact that each physical device access requires the scheduling and execution of multiple domains. The SV-IO abstraction facilitates alternative, *device-centric*, realizations that address this issue, using metrics that include both the scalability of virtualization and the raw performance of individual virtualized devices. *A device-centric SV-IO realization implements selected virtualization functionality on the device itself*, resulting in less host involvement and potential performance benefits.

To demonstrate the utility of the device-centric SV-IO realization, we have created a self-virtualized network device (SV-NIC) on an implementation platform comprised of an

IA-based host and an IXP2400 network processor-based gigabit ethernet board, using the Xen HV [48]. Since the IXP2400 network processor contains multiple processing elements situated *close* to the physical I/O device, this device-centric SV-IO realization efficiently exploits these resources to offer levels of performance exceeding that of the host-centric realizations used in existing systems. In particular, the self-virtualized network device (SV-NIC):

1. exploits IXP-level resources by mapping substantial virtualization functionality (*i.e.*, the HV functions and the device stack required to virtualize the device) to multi-core resources located *near* the physical network device,

2. removes most HV interactions from the data fast path, by permitting each guest domain to directly interact with the virtualized device, and

3. avoids needless transitions across the (relatively slow) PCI-based communication link between host and device.

The implementation also frees host computational resources from simple communication tasks, thereby permitting them to be utilized by guest VMs and improving platform scalability to permit a larger number of virtual machines.

The scalability of device virtualization solutions constructed with the SV-IO abstraction depends on two key factors: (1) the virtual interface (VIF) abstraction and its associated API, and (2) the algorithms used to manage multiple virtual devices. For the SV-NIC, measured performance results show it to be highly scalable in terms of resource requirements. Specifically, limits on scalability are not due to the design but are dictated by the availability of resources at the discretion of SV-IO, such as the physical communication bandwidth and the maximum number of processing cores that can be deployed by the SV-IO. Results also show how certain design choices made in the SV-NIC implementation of SV-IO are affected by, and/or suggest the utility of, specific architectural features of modern computing platforms. One example is the necessity of integrating I/O MMU support with SV-IO.

The purpose of SV-IO support for both device- and host-centric implementations of device virtualization is to give system developers the flexibility to make choices suitable

for specific target platforms. Factors to be considered in such choices include actual host *vs.* device hardware, host- *vs.* device-level resources, the communication link between them, and system and application requirements. In fact, evidence exists for both host- and device-centric solutions. The former represents a current industry trend that aims to exploit general multi-core resources. The latter is bolstered by substantial prior research, with examples including intelligent network devices [117, 70, 152], disk subsystems [116, 11], and even network routers [145], with recent work focusing on network virtualization [9].

Performance results demonstrate that the SV-IO abstraction meets its joint goals of high performance and flexibility in implementation. For a platform with a high end network device, for example, we show that the device-centric realization of SV-IO results in an SV-NIC that permits virtual devices to operate at full link speeds for 100 Mbps ethernet links. At gigabit link speeds, PCI performance dominates the overall performance of virtual devices. A VIF from this SV-NIC provides TCP throughput and latency of $\sim$ 620Mbps and $\sim$ .076ms, respectively, which is $\sim$77% more throughput and $\sim$53% less latency when compared to a VIF from a *host-centric* SV-IO realization. Finally, performance scales well for both host- and device-centric SV-IO realizations with an increasing number of virtual devices, one device per guest domain, although the device-centric realization performs better. For example, for 8 VIFs, the aggregate throughput (latency) for the device-centric version is 103% more (39% less) compared to the host-centric version.

2.2 The SV-IO Abstraction

Self-virtualized I/O (SV-IO) is a *hypervisor-level abstraction designed to encapsulate the virtualization of I/O devices*. Its goals are:

- scalable multiplexing/demultiplexing of a large number of *virtual devices* mapped to a single physical device,
- providing a lightweight API to the HV for managing virtual devices,
- efficiently interacting with guest domains via simple APIs for accessing the virtual devices, and

- harnessing the compute power (*i.e.*, potentially many processing cores) offered by future hardware platforms.

Before we describe the different components of the SV-IO abstraction and their functionalities, we briefly digress to discuss the virtual interface (VIF) abstraction provided by SV-IO and the associated API for accessing a VIF from a guest domain.

2.2.1 Virtual Interfaces (VIFs)

Examples of *virtual I/O devices* on virtualized platforms include virtual network interfaces, virtual block devices (disk), virtual camera devices, and others. Each such device is represented by a *virtual interface* (VIF) which exports a well-defined interface to the guest OS, such as ethernet or SCSI. The virtual interface is accessed from the guest OS via a VIF device driver.

Each VIF is assigned a *unique ID*, and it consists of two message queues, one for outgoing messages to the device (*i.e.*, *send queue*), the other for incoming messages from the device (*i.e.*, *receive queue*). The simple API associated with these queues is as follows:

```
boolean isfull(send queue);
size_t send(send queue, message m);
boolean isempty(receive queue);
message recv(receive queue);
```

The functionality of this API is self-explanatory.

A pair of signals is associated with each queue. For the send queue, one signal is intended for use by the guest domain, to notify the SV-IO that the guest has enqueued a message in the send queue. The other signal is used by the SV-IO to notify the guest domain that it has received the message. The receive queue has signals similar to those of the send queue, except that the roles of guest domain and SV-IO are interchanged. A particular implementation of SV-IO need not use all of these defined signals. For example, if the SV-IO polls the send queue to check the availability of message from the guest domain, it is not required to send the signal from guest domain to the SV-IO. Furthermore, queue

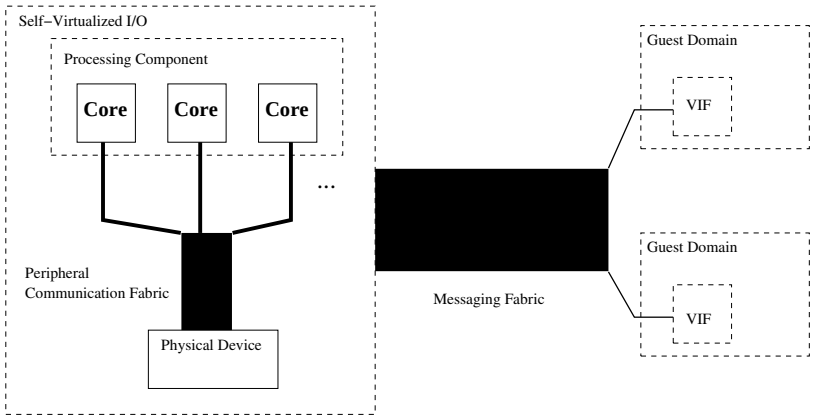


Figure 2: SV-IO abstraction.

signals are configurable at runtime, so that they are only sent when expected/desired from the other end. For example, a network driver using NAPI [120] does not expect to receive any interrupts when it processes the receive for a bunch of incoming network packets.

2.2.2 SV-IO Design

The SV-IO abstraction has four logical components, as depicted in Figure 2. The *processing component* consists of one or more *cores*. This component is connected to the *physical I/O device* via the *peripheral communication fabric*. Guest domains communicate with the SV-IO using VIFs and via the *messaging fabric*.

The two main functions of SV-IO are *managing VIFs and performing I/O*. Management involves creating or destroying VIFs or reconfiguring various parameters associated with them. These parameters define VIF performance characteristics, and in addition, they can be used by guest domains to specify QoS requirements for the virtual device. When performing I/O, in one direction, a message sent by a guest domain over a VIF's send queue is received by the SV-IO's processing component. The processing component then performs all required processing on the message and forwards it to the physical device over the peripheral communication fabric. Similarly, in the other direction, the physical device sends data to the processing component over the peripheral communication fabric, which then demultiplexes it to one of the existing VIFs and sends it to the appropriate guest

domain via the VIF's receive queue. A key task in processing message queues is for SV-IO to multiplex/demultiplex multiple VIFs on a single physical I/O device. The scheduling decisions made as part of this task must enforce performance isolation among different VIFs. While there are many efficient methods for making such decisions, *e.g.* DWCS [144], the simple scheduling method used in experimentation presented in this chapter is round-robin scheduling.

2.3 Realizing SV-IO: Design Choices

Modern computing platforms' rich architectural resources provide many design choices when realizing components of the SV-IO abstraction:

- The peripheral communication fabric connecting the processing component to the physical device could be a dedicated/specialized interconnect, such as the Media and Switch Fabric (MSF) that is used to connect the IXP2400's network processor cores to the physical network port [78], or it could be a shared interconnect like PCI or HyperTransport.

- The messaging fabric connecting the processing component to guest domains could be realized using shared memory (with/without coherence), an interconnect like PCI, or a combination thereof, depending on the locations of *cores* in the SV-IO's processing component.

- Cores in the processing component could be heterogeneous or homogeneous, in contrast to the homogeneous cores used by guest domains. For example, rather than using all IA32-based cores as done by the guest domains, SV-IO could be mapped to specialized processor cores designed for network I/O processing. An advantage of specialized cores is that they need not offer the multitude of resources and features present in general cores, thereby saving chip real-estate while still providing comparable or even improved performance [85]. Another advantage is improved platform power efficiency. Potential disadvantages of heterogeneous cores are well known. One is the cost of implementing I/O subsystem software on platforms with different instruction

sets and requiring different programming methods. Another is their comparative inflexibility compared to general cores, making it difficult to implement more complex functionality [152] there.

- Another choice for the processing component is to use dedicated cores to realize SV-IO, or to multiplex such I/O functions on cores shared with other processing activities. For high performance systems, there is evidence of performance advantages, due to reduced OS 'noise', derived from at least temporally dedicating certain cores to carry out I/O *vs.* computational tasks [107].

2.3.1 SV-IO Implementations

Our SV-IO implementations utilize the Xen hypervisor. In Xen, the standard implementation of device I/O uses *driver domains*, which are special guest domains that are given direct access to physical devices via some physical interconnect (*e.g.*, PCI). The driver domain provides the virtual interfaces, *e.g.*, a virtual block device or a virtual network interface, to other guest domains. The driver domain also implements the multiplex/demultiplex logic for sharing the physical device among virtual interfaces, the logic of which depends on the properties of each physical device. For instance, time sharing is used for the network interface, while space partitioning is used for storage. The hypervisor schedules the driver domains to run on general purpose host cores.

While Xen is not currently structured using SV-IO, the functions it runs in the driver domain for each physical device being virtualized are equivalent to those of an SV-IO-based device. Host cores belonging to the driver domain are the SV-IO's processing components, and they are architecturally homogeneous to the cores running guest domains. Host cores also run the SV-IO components that provide its management and I/O functionality. The peripheral communication fabric is implemented via the peripheral interconnect, *e.g.*, PCI. The messaging fabric to communicate between cores running the driver domain and guest domains is implemented via shared memory. The sharing of cores used by the processing component is dependent on the hypervisor's scheduling policy. We refer to this approach as a *host-centric* realization of SV-IO, since all virtualization logic executes on host cores.

A *device-centric* realization of SV-IO exploits the processing elements 'close to' physical devices, such as processing elements in network processor-based platforms [6] and in SCSI adapters [11]. In a device-centric realization, the interconnect between the physical I/O device and on-device processing elements form the peripheral communication fabric, *e.g.* MSF, while the interconnect between the host system and the high end I/O device forms the messaging fabric, *e.g.*, PCI. Performance and/or scalability for the SV-IO device are improved when it is possible to better exploit the device's processing resources, to improve device behavior due to 'fabric near' control actions [117], or to shorten the path from device to guest domain. A specific example of a device-centric SV-IO realization is presented in the next section.

We choose the terms *device- or host-centric to refer to the location(s) of the majority rather than the entirety of SV-IO processing functionality.* Our SV-NIC implementation, for instance, requires host assistance for certain control plane device/guest interactions. Similarly, host-centric SV-IO will require some degree of device-level support, *e.g.*, the capability to perform I/O.

2.4 SV-IO Realizations for High-End Network Interface Virtualization

2.4.1 Hardware Platform and Basic Concepts

The communication device used is an IXP2400 network processor(NP)-based RadiSys ENP2611 board [6]. This resource-rich network processor features a XScale processing core and 8 RISC-based specialized communication cores, termed *micro-engines.* Each micro-engine supports 8 hardware contexts with minimal context switching overhead. The physical network device on the board is a PM3386 gigabit ethernet MAC connected to the network processor via the *Media and Switch Fabric* (MSF) [78]. The board also contains substantial memory, including SDRAM, SRAM, scratchpad and micro-engine local memory (listed in the order of decreasing sizes and latencies, and increasing costs.) The board runs an embedded Linux distribution on the XScale core, which contains, among others, some management utilities to execute *micro-code* on the micro-engines. This micro-code is the sole execution entity that runs on the micro-engines.

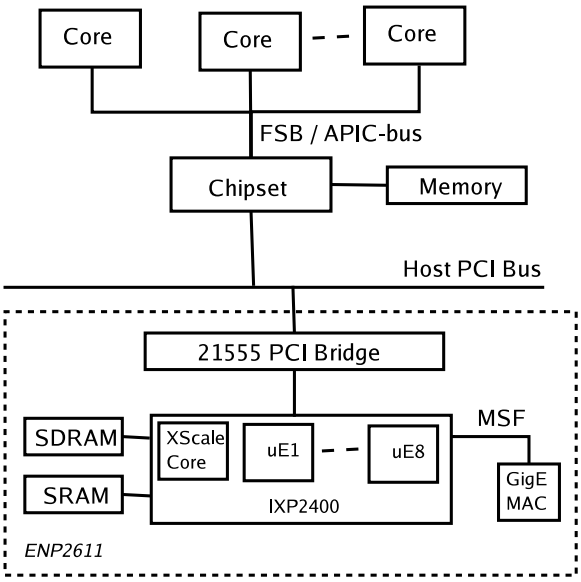


Figure 3: Host-NP platform.

The combined host-NP platform represents one point in the design space of future multi-core systems, offering heterogeneous cores for running applications, guest OSes, and I/O functionality. As will be shown later, the platform is suitable for evaluating and experimenting with the scalability and with certain performance characteristics of the SV-IO abstraction, but it lacks the close coupling between host and NP resources likely to be found in future integrated multi-core systems. Specifically, in our case, the NP resides in the host system as a PCI add-on device, and it is connected to the host PCI bus via the Intel 21555 non-transparent PCI bridge [14]. This bridge allows the NP to only access a portion of host RAM resources via a 64MB PCI address window. In contrast, in the current configuration, host cores can access all of the NP's 256MB of DRAM.

The following details about the PCI bridge are relevant to some of our performance results. The PCI bridge contains multiple *mailbox* registers accessible from both host- and NP-ends. These can be used to send information between host cores and NP. The bridge also contains two interrupt identifier registers called *doorbell*, each 16-bit wide. The NP can send an interrupt to the host by setting any bit in the host-side doorbell register. Similarly, a host core can send an interrupt to the *XScale core* of the NP by setting any bit in the NP-side doorbell register. Although the IRQ asserted by setting bits in these registers is the

same, the IRQ handler can differentiate among multiple "reasons" for sending the interrupt by looking at the bit that was set to assert the IRQ.

2.4.2 Host-Centric Implementation of SV-IO

As explained in the previous section, in a host-centric realization of SV-IO, the network interface's virtualization logic runs in the driver domain (or controller domain) on host cores. The processing power available on the NP is used to tunnel network packets between the host and the gigabit ethernet interface residing on the board. This provides to the host the illusion that the ENP2611 board is a gigabit ethernet interface. In fact, this tunnel interface is almost identical to a VIF. It contains two queues, a send-queue and a receive-queue, and it bears the ID of the physical ethernet interface. These queues contain a ring structure for queue maintenance and the actual packet buffers.

The NP's XScale core is not involved in the data fast path. Its role is to carry out control actions, such as starting and stopping the NP's micro-engines. The data fast path, *i.e.*, performing network I/O, is solely executed by micro-engines. In particular, a single micro-engine thread polls the send-queue and sends out packets queued by the device driver running in the driver domain onto the physical port. In case the driver domain fills up the entire queue before the micro-engine thread services it, the driver domain requests a signal to be sent when further space in the send-queue is available. The micro-engine thread sends this signal after it has processed some packets from the send-queue. A second micro-engine's execution contexts are used for receive-side processing – they select the packets from the physical interface and enqueue them on the receive-queue, in order. For each packet enqueued, a signal is sent to the driver domain, if required. The host side driver for the tunnel interface uses NAPI, which may disable this signal to reduce the signal processing load on the host in case the packet arrival rate is high. Thus, *the signals are only sent by the NP to driver domain.* Both signals are implemented as different identifier bits of the host-side doorbell register; the IRQ handler running in the driver domain determines the type of signal based on the identifier bit.

Software ethernet bridging, virtual network interfaces, front-end device drivers in guest

domains, and back-end device drivers in the driver domain are used to virtualize this tunnel device. Xen's network interface virtualization is described in detail in [110].

2.4.3 SV-NIC: Device-Centric Implementation

In our NP-based, device-centric implementation, termed SV-NIC, most of the processing component, the peripheral communication fabric, and the physical I/O device components of the SV-IO abstraction are situated on the ENP2611 board itself. The board's programmability and substantial processing resources make it easy to experiment with alternative SV-NIC implementation methods, as will become evident in later sections.

The SV-NIC implementation uses the following mapping for SV-IO components. The processing component is mapped to the XScale core and the micro-engines available on the board, along with one or more host processing cores. The peripheral communication fabric consists of the *Media and Switch Fabric* (MSF) [78]. The physical I/O device, the PM3386 gigabit ethernet controller, connects to the network processor via MSF. The processing component uses PCI as the messaging fabric to communicate with the guest domains via the virtual interface (VIF) abstraction.

The SV-NIC directly exports its VIF abstraction to guest domains as virtual network device. The send queue of each VIF is used for outgoing packets from guest domains and the receive queue is used for incoming packets to guest domains. As explained in more detail in the next subsection, only some of the signals associated with VIFs are needed: those sent from the SV-NIC to the guest domain. These two signals work as transmit and receive interrupts, respectively, similar to what is needed for physical network devices. Both signals are configurable and can be disabled/enabled at any time by the guest domain virtual interface driver, as required. For example, the send code of the guest domain driver does not enable the transmit interrupt signal till it finds that the send queue is full (which will happen if SV-IO is slower than the host processor). Similarly, the receive code of the guest domain driver uses the NAPI interface and disables receive interrupt signal when processing a set of packets. This reduces the interrupt load on the host processor when the rate of incoming packets is high. The queues have configurable sizes that determine

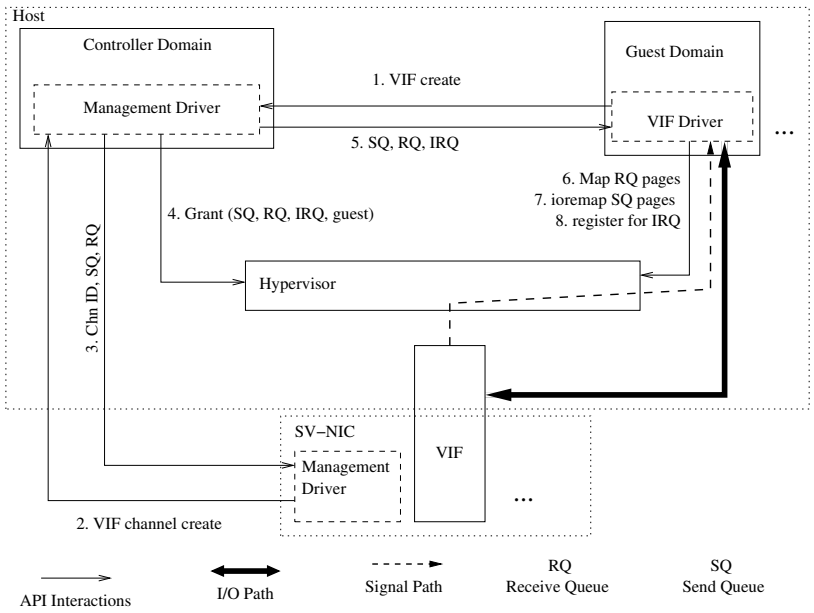


Figure 4: Management interactions between SV-NIC, hypervisor and the guest domain to create a VIF. Shaded region depicts the boundary of SV-IO abstraction.

transmit and receive buffer lengths for the store and forward style communication between SV-NIC and guest domain.

2.4.3.1 Functionality breakdown of processing components for SV-NIC

This section describes how the cores used for the processing component of the SV-NIC achieve (1) VIF management and (2) network I/O.

Management functionality includes the creation of VIFs, their removal, and changing attributes and resources associated with them. Figure 4 depicts various *management interactions* between the SV-NIC's processing components and the guest domain to *create a VIF*. The figure also shows the I/O and signaling paths for the VIF between the SV-NIC and the guest domain (via the messaging fabric). Setup and usage of these paths is deferred to Section 2.4.3.2, since it is dependent on various techniques employed by the Xen hypervisor.

Other management functionality includes the *destruction of VIF* and *changing attributes* of a VIF or of SV-NIC. Destruction requests are initiated by the hypervisor when a VIF has to be removed from a guest. This might be the result of a guest VM shutdown, or for security reasons (*e.g.* when a VM is compromised, its NICs can be torn apart.)

Certain *attributes* can be set at VIF creation time or later to change VIF properties. For example, the throughput achievable by a VIF directly depends on the *buffer space* provided for the send- and receive-queues. Throughput and latency also depend on the *scheduling algorithm* used at the NP for the processing of packets corresponding to different VIFs. Hence, changing these attributes will affect runtime changes in VIF behavior.

Management functionality is accomplished by two management drivers that execute on different processing components of the SV-NIC. The host-side driver is part of the OS running in the controller domain (Dom0). It runs on the host core(s). The device-side driver is part of the embedded OS running on the NP-based board. It runs on the XScale core.

Management requests are generated by guest domains or the hypervisor. They are forwarded to the host-side management driver, which in turn forwards relevant parameters to the device-side driver via the 21555 bridge's mailbox registers. The device-side driver appropriates the resources for VIFs, which includes assigning micro-engines for network I/O and messaging fabric space for send/receive queues. The device-side driver then communicates these changes to the host-side driver, via the bridge's mailbox registers, and to the micro-code running on the micro-engines, via SRAM.

A guest domain performs network I/O via a VIF. It enqueues packets on the VIF's send-queue and dequeues packets from the VIF's receive-queue. It is the responsibility of the SV-NIC to:

- *egress:* multiplex packets in the send-queues of all VIFs on to the physical device; and
- *ingress:* demultiplex the packets received from the physical network device onto appropriate VIFs' receive queues.

Since VIFs export a regular ethernet device abstraction to the host, this implementation models a software layer-2 (ethernet) switch.

In our current implementation, *egress is managed by one micro-engine context per VIF.* For simple load balancing, this context is selected from a pool of contexts belonging to a single micro-engine (the egress micro-engine) in a round robin fashion. Hence, the lists of

VIFs being serviced by the contexts of the egress micro-engine are mutually disjoint. This allows for lock free operation of all contexts. The contexts employ *voluntary* yielding after processing every packet and during packet processing for I/O, to maintain a fair-share of physical network resources across multiple VIFs.

Ingress is managed for all VIFs by a shared pool of contexts belonging to one micro-engine (the ingress micro-engine). Each context selects a packet from the physical network, *demultiplexes* it to a VIF based on MAC address, locks the VIF, obtains a *sequence number* to perform "in-order" placement of packets, unlocks the VIF, and signals the *next* context to run. Next it performs the I/O action of moving the packet to the VIF receive-queue, during which it voluntarily relinquishes the micro-engine to other contexts that are either performing I/O or waiting for a signal from the *previous* context in order to get a chance to execute. After a context is done performing I/O, it waits for the expected sequence number of the VIF to match its sequence number, yielding the micro-engine voluntarily between checking for this condition to become true. Once this wait operation is complete, the context atomically adjusts the VIF's receive-queue data structures to signify that a packet is successfully queued. Also, a signal to the guest domain is sent if required by the guest domain driver.

Our SV-NIC sends signals to the guest domain, and its micro-engines poll for information from the guest domains. There are multiple reasons for this design: (1) an ample number of micro-engines and fast switchable hardware contexts make it cheaper to poll for information than to wait for an asynchronous signaling mechanism like an interrupt; (2) hardware contexts running on micro-engines are non-preemptible, thus the context must explicitly check for the presence of interrupt signal anyway; and (3) there exists no direct signaling path from host cores to micro-engines, so that such signals would have to be routed via the XScale core, resulting in prohibitive latency.

More specifically, every VIF is assigned two different bits in the host-side interrupt identifier register (one each for the send and receive directions). The bits are shared by multiple VIFs in case the total number of VIFs exceeds 8. Setting any bit in the identifier register causes a master PCI interrupt to be asserted on the host core(s) of SV-IO's processing

component. Using the association between bits and VIFs, the SV-NIC can determine which VIF (or *potential set of VIFs* in case of sharing) generated the master interrupt, along with the reason, by reading the identifier register. Based on the reason (send/receive), an appropriate signal is sent to the guest domain associated with the VIF(s). This signal demultiplexing functionality of SV-IO is implemented as part of the Xen hypervisor itself.

In our current implementation, the master PCI interrupt generated by SV-NIC is sent to a specific host core. This core runs the signal demultiplexing and forwarding (aka interrupt virtualization) in hypervisor context. Thus, the set of host cores, which is a part of SV-IO's processing component, includes the cores assigned for the controller domain and the core performing interrupt virtualization.

2.4.3.2 Management Role of the Xen HV

Our device-centric realization of SV-IO, the SV-NIC, provides VIFs directly to guest domains. There is *minimal* involvement of the HV, and little additional host-side processing is required for I/O virtualization. The previous section described the interrupt virtualization role played by the HV for the SV-NIC. This section describes the management role of the HV in the setup phase of a VIF.

In order for a guest domain to utilize the VIF provided by the SV-NIC, it must be able to:

- *write messages* in the NP SDRAM corresponding to the *VIF send queue*; and
- *read messages* from the host RAM corresponding to the *VIF receive queue*.

The NP's SDRAM is part of the host PCI address space. Access to it is available by default only to privileged domains, *e.g.*, the controller domain. In order for a (non-privileged) guest domain to be able to access its VIF's send queue in this address space, the management driver uses Xen's *grant table mechanism* to authorize write access to the corresponding I/O memory region for the requesting guest domain. The guest domain can then request Xen to map this region into its page tables. Once the page table entries are installed, the guest domain can inject messages directly into the send queue. For security

reasons, the ring structure part of this region is read-only mapped for the guest, while the other part containing the packet buffers is mapped read-write. This is necessary because if the ring structure was writable, a malicious guest could influence the NP to read from arbitrary locations and inject bogus packets on the network.

In our current implementation, the host memory area accessible to the NP is owned by the controller domain. The management driver grants access of the region belonging to a particular VIF to its corresponding guest domain. The guest domain then asks Xen to map this region into its page tables and can subsequently receive messages directly from the VIF's receive queue. The part of this region containing the ring structure is mapped read-only, while the part containing actual packet buffers is mapped read-write. The above mappings are created once during VIF creation time and remain in effect for the life-time of the VIF (usually the life-time of its guest domain). All remaining logic to implement packet buffers inside the queues and the send/receive operations is implemented completely by the guest domain driver and on the NP micro-engines.

The ring structure is mapped read-only so that a malicious guest cannot influence the NP to perform writes to memory areas it does not own, thereby corrupting other domain's state. This is similar to the issue of DMA security isolation: if a guest domain is allowed to program DMA addresses in a device, then it can program it to an area of memory that it may not own, thereby corrupting the hypervisor's or other domain's memory. In Section 2.5.2, we describe how this issue can be addressed with future hardware I/O MMUs.

In summary, the grant table mechanism described above enforces security isolation – a guest domain cannot access the memory space (neither upstream nor downstream) of VIFs other than its own. Also, since a guest domain cannot perform any management related functionality, it cannot influence the NP to perform any illegal I/O to a VIF that it does not own.

2.5 Platform-Specific Implementation Details and Insights

In this section, we discuss some platform-specific limitations and how our current SV-IO realizations deal with them, along with insights on improvements possible with certain

enhancements.

2.5.1 PCI Performance Limitations

Key requirements for the virtualized network device are (1) low communication overhead and (2) high performance in terms of bandwidth and latency. To attain both, our VIF and tunnel device implementations must deal with certain limitations of our chosen host/NP implementation platform. One challenge is to deal with the platform's limitations on PCI read bandwidth, as shown by a microbenchmark in Section 2.6.3.2. Toward this end, the current implementation avoids PCI reads whenever possible, by placing the send message queue into the NP's SDRAM (the *downstream communication space*), while the receive message queue is implemented in host memory (the *upstream communication space*). As a result, both on egress and ingress paths, PCI reads are avoided by guest domains and by SV-IO's processing components since relevant information is available in *local* memory.

2.5.2 Need for I/O MMU

Unlike the case of downstream access, where the host can address any location in the NP's SDRAM, current firmware restrictions limit the addressability of host memory by the NP to 64MB. Even with firmware modifications, the hard limit is 2GB. Since the NP cannot access the complete host address space, all NP to host data transfers must target specific buffers in host memory, termed *bounce buffers*. The receive queue of the tunnel device or a VIF consists of multiple bounce buffers. For ease of implementation, all bounce buffers are currently allocated contiguously in host memory, but this is not necessary with this hardware.

In keeping with standard Unix implementations, the host-side driver copies the network packet from the bounce buffer in the receive queue to a socket buffer (*skb*) structure. An alternate approach avoiding this copy is to directly utilize receive queue memory for *skb*s. This can be achieved by either (1) implementing a specific skb structure and a new page allocator that uses the receive queue pages, or (2) instead of having a pre-defined receive queue, construct one that contains the *bus addresses* of allocated skbs. The latter effectively requires either that the NP is able to access the entire host memory (which is not possible

due to the limitations discussed above) or that an I/O MMU is used for translating a bus address accessible to the NP to the memory address of the allocated skb. For ease of implementation, we have not pursued (1) in our prototype, but it is an optimization we plan to consider in future work. Concerning (2), since our platform does not have a hardware I/O MMU, our implementation emulates this functionality by using bounce buffers plus message copying, essentially realizing a software I/O MMU. In summary, *the construction of the network I/O path would be facilitated by efficient upstream translated NP accesses to host memory.*

A related issue for the SV-NIC is to provide performance and security isolation among multiple VIFs. Our implementation attains performance isolation on the NP itself by spatially partitioning memory resources and time-sharing the NP's micro-engine hardware contexts. Some aspect of security isolation is provided by the host hypervisor, as discussed in the previous section. We next discuss the role of I/O MMUs in the context of security isolation.

For security isolation in the upstream network I/O path, we rely on the fact that the ring structure in the receive queue of a VIF is immutable to the guest. This requirement can be relieved by having the NP perform run-time checks to ensure that the bus address provided by the guest on the receive queue ring refers to a memory address that indeed belongs to the guest domain. These runtime checks can remove the need for bounce buffers in case the device can access all host memory, which may be facilitated by a hardware I/O MMU. Performing these checks is straightforward when the hypervisor statically partitions the memory among the guest domains, since it reduces to a simple range check. However, a domain can have access to certain memory pages that are "granted" to it by other domains, thereby implying that a runtime check must also search the grant table of the guest domain owning the VIF. Another approach would be for the hypervisor to provide a *map*, or bit vector of all of the memory pages currently owned by a guest domain. Based on this map, either the self-virtualized device or the hardware I/O MMU can decide whether an upstream I/O transaction takes place, depending on whether the target bus address is owned by the guest domain. Recent I/O MMUs available with hardware-assisted virtualization

technology, such as AMD's Pacifica [4] or Intel's VT-D [43], support similar functionality, albeit for providing exclusive access of a single I/O device to a guest domain. These I/O MMUs must be enhanced to include multiple virtual devices per physical device in order to be useful with self-virtualized devices.

2.6 Performance Evaluation

2.6.1 Experiment Basis and Description

The experiments reported in this chapter use two hosts, each with an attached ENP2611 board. The gigabit network ports of both boards are connected to a gigabit switch. Each host has an additional Broadcom gigabit ethernet card, which connects it to a separate subnet for developmental use.

Hosts are dual 2-way HT Pentium Xeon (a total of 4 logical processors) 2.80GHz servers, with 2GB RAM. The hypervisor used for system virtualization is Xen3.0-unstable [110]. Dom0 runs a paravirtualized Linux 2.6.16 kernel with a RedHat Enterprise Linux 4 distribution, while guest VMs run a paravirtualized Linux 2.6.16 kernel with a small ramdisk root filesystem based on the Busybox distribution. The ENP2611 board runs a Linux 2.4.18 kernel with the MontaVista Preview Kit 3.0 distribution. Experiments are conducted with uniprocessor Dom0 and guest VMs. Dom0 is configured with 512MB RAM, while each guest VM is configured with 32MB RAM. We use the default Xen CPU allocation policy, under which Dom0 is assigned to the first hyperthread of the first CPU (logical CPU #0), and guest VMs are assigned one hyperthread from the second CPU (logical CPU #2 and #3). Logical CPU #1 is unused in our experiments. The Borrowed Virtual Time (bvt) scheduler with default arguments is the domain scheduling policy used for Xen.

Experiments are conducted to evaluate the costs and benefits of host- *vs.* device-centric SV-IO realizations, for virtualized hosts. For the sake of brevity, we nickname these realizations HV-NIC and SV-NIC, respectively. Two sets of experiments are performed. The first set uses HV-NIC, where the driver domain provides virtual interfaces to guest domains. Our setup uses Dom0 (*i.e.*, the controller domain) as the driver domain. Using the host-centric approach as the base case, the second set of experiments evaluates the SV-NIC realization

described in Section 2.4.3, which provides VIFs directly to guest domains without any driver domain involvement in the network I/O path.

The performance of SV-NIC *vs.* HV-NIC realizations, *i.e.*, of their virtual interfaces provided to guest domains, are evaluated with two metrics: latency and throughput. For latency, a simple libpcap [31] client server application, termed *psapp*, is used to measure the packet round trip times between two guest domains running on different hosts. The client sends 64-byte probe messages to the server using packet sockets and SOCK_RAW mode. These packets are directly handed to the device driver, without any Linux network layer processing. The server receives the packets directly from its device driver and immediately echoes them back to the client. The client sends a probe packet to the server and waits indefinitely for the reply. After receiving the reply, it waits for a random amount of time, between 0 and 100ms, before sending the next probe. The RTT serves as an indicator of the inherent latency of the network path.

For throughput, we use the iperf [19] benchmark application. The client and the server processes are run in guest VMs on different hosts. The client sends data to the server over a TCP connection with buffer size set to 256KB (on guest VMs with 32MB RAM, Linux allows only a maximum of 210KB), and the average throughput for the flow is recorded. The client run is repeated 20 times.

All experiments run on two hosts and use a 'n,n:1x1' access pattern, where 'n' is the number of guest domains on each host. Every guest domain houses one VIF. On one machine, all guest domains on a machine run server processes, one instance per guest. On the second machine, all guest domains run client processes, one instance per guest. Each guest domain running a client application communicates to a distinct guest domain that runs a server application on the other host. Hence, there are a total n simultaneous flows in the system. In the experiments involving multiple flows, all clients are started simultaneously at a specific time in pre-spawned guest domains. We assume that the time in all guest domains is kept well-synchronized by the hypervisor (with resolution at 'second' granularity).

2.6.2 Experimental Results

2.6.2.1 Latency

The latency measured as the RTT by the psapp application includes both basic communication latency and the latency contributed by virtualization. Virtualization introduces latency in two ways: (1) a packet must be classified as to which VIF it belongs to, and (2) the guest domain owning this VIF must be notified. Based on the MAC address of the packet and using hashing, classification can be done in constant time for any number of VIFs, assuming no hash collision.

For the HV-NIC, step (2) above requires sending a signal from the driver to the guest domain. This takes constant time, but with increasing CPU contention, additional end-to-end latency would be caused if a target guest were not immediately scheduled to process the signal. Thus, with an increasing number of VIFs, we would expect latency values to increase and exhibit larger variances. Finally, since the driver domain and guest domains are scheduled on different CPUs, sending a signal to a guest domain involves an IPI (inter-processor interrupt).

For the SV-NIC, step (2) above requires the hypervisor to virtualize the PCI interrupt and forward it as a signal to the guest domain, as described in Section 2.4.3. In case the host core responsible for interrupt virtualization is being shared by the target guest domain, sending this signal is done via a simple upcall, which is cheaper than performing an IPI. Given that all guest domains are scheduled on two CPUs, on the average, signal forwarding from one of these two cores provides the performance optimization 50% of the time. As with the HV-NIC case, if multiple guest domains are sharing a CPU, the target guest domain may not be scheduled right away to process the signal sent by the self-virtualized network interface. Thus, with an increasing number of VIFs, we would expect latency values to increase and exhibit larger variances.

Another source of latency in device-centric case is the total number of signals that need to be sent per packet. As mentioned earlier in Section 2.4.3.1, due to the limitations on interrupt identifier size, a single packet may require more than one guest domains to be signaled. In particular, the total number of domains signaled, n_s, is given by the following

formula: $\begin{cases} \lfloor n/l \rfloor \leq n_s \leq \lceil n/l \rceil & \text{if } n > l \\ 1 & \text{otherwise} \end{cases}$, where n is the total number of domains and l is the interrupt identifier size. Thus, the smaller the l, the more the number of domains that will be signaled (all but one of which would be redundant), and vice versa. Assuming these domains share the CPU, the overall latency will include the time it takes to send a signal and *possibly* the time spent for useless work performed by a redundant domain; latter of which will be decided by domain scheduling on the shared CPU.

Using RTT as the measure of end-to-end latency, Figure 5 shows the RTT reported by *psapp* for HV-NIC and SV-NIC. On the x-axis is the total number of concurrent guest domains 'n' running on each host machine. On the y-axis is the *median* latency and inter-quartile range of the 'n' concurrent flows; each flow $i \in n$ connects $GuestDomain_i^{client}$ to $GuestDomain_i^{server}$. For each n, we combine N_i latency samples from flow $i, 1 \leq i \leq n$ as one large set containing $\sum_{i=1}^{n} N_i$ samples. The reason is that each flow measures *the same* random variable, which is end-to-end latency when n guest domains are running on both sides.

We use the median as a measure of central tendency since it is more robust to outliers (which occur sometimes due to unrelated system activity, especially under heavy load with many guest domains.) Inter-quartile range provides an indication of the spread of values.

These results demonstrate that with the device-centric approach to SV-IO, it is possible to obtain close to a 50% latency reduction for VIFs compared to Xen's current host-centric implementation. This reduction results from the fact that Dom0 is no longer involved in the network I/O path. In particular, the cost of scheduling Dom0 to demultiplex the packet, using bridging code, and sending this packet to the front-end device driver of the appropriate guest domain is eliminated on the receive path. Further, the cost of scheduling Dom0 to receive a packet from guest domain front-end and to determine the outgoing network device using bridging code is eliminated on the send path. Also, with SV-NIC, the latency of using one of its VIFs in a guest VM is almost identical to using the tunnel interface from the domain that has direct device access, Dom0. Our conclusion is that the basic cost of the device-centric implementation is low. Also demonstrated by these measurements is that the

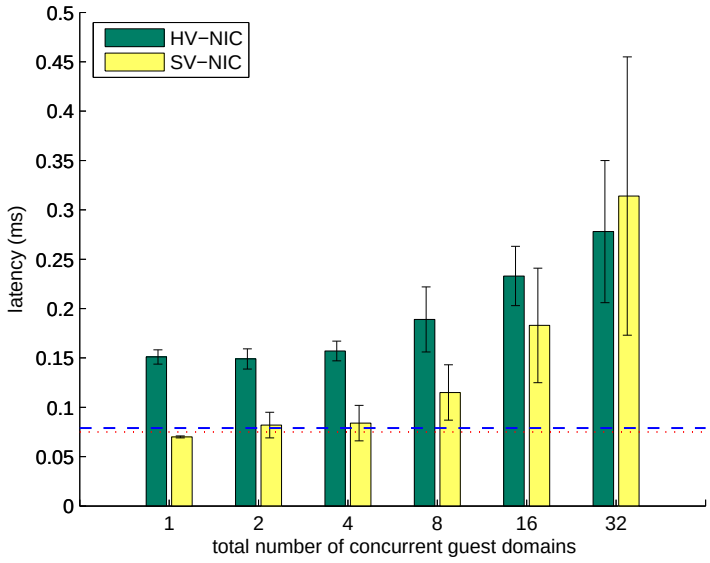


Figure 5: Latency of HV-NIC and SV-NIC. Dotted lines represent the latency for Dom0 using the tunnel network interface in two cases: (1) No SV-IO functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric SV-IO functionality (*i.e.*, with software bridging), represented by dash dots.

cost of our SV-NIC implementation is fully contained in the device and the HV.

The median latency value and inter-quartile range increases in all cases as the number of guest domains (and hence the number of simultaneous flows) increases. This is mostly because of increased CPU contention between guests. Also, due to interrupt identifier sharing, the latency of SV-NIC increases beyond that of HV-NIC for 32 VIFs. In that case, every identifier bit is shared among 4 VIFs, and hence, requires 1.5 redundant domain schedules on the average before a signal is received by the correct domain. On our system with only two CPUs available for guest VMs, these domain schedules also require context switching, which further increases latency.

Since latency degrades due to CPU contention among guests, we expect to see better performance if a large, *e.g.* a 32-way, SMP were used. In that case, SV-NIC would perform better for 32 guests even though the virtual interrupt identifier is shared. This is because all signaled domains would be running on different CPUs. The performance of the HV-NIC remains similar to the case of the single guest domain for a 2-way SMP (results for which are shown in Figure 5.)

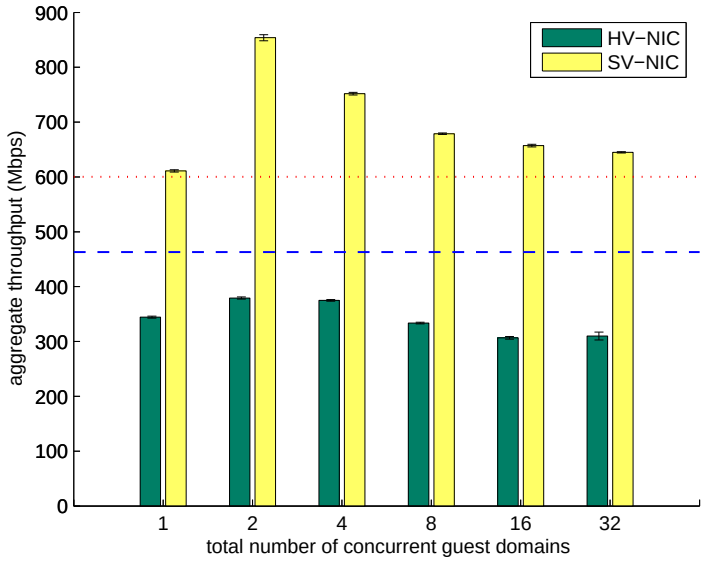


Figure 6: TCP throughput of HV-NIC and SV-NIC. Dotted lines represent the throughput for Dom0 using tunnel network interface in two cases: (1) No SV-IO functionality (*i.e.*, without software bridging), represented by fine dots, and (2) host-centric SV-IO functionality (*i.e.*, with software bridging), represented by dash dots.

2.6.2.2 Throughput

The aggregate throughput achieved by n flows is the sum of their individual throughputs. Particularly, if we denote the aggregate throughput for n flows as a random variable T, it will be equal to $\sum_{i=1}^{n} T_i$, where T_i denotes the random variable corresponding to the throughput for flow i. Since we expect each T_i to have finite mean μ_i and variance σ_i^2, and since they are independent of each other (they may not be normally distributed), we expect T to follow a normal distribution $N(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \sigma_i^2)$ according to the central limit theorem. We estimate the mean and variance for each flow by the sample mean and variance.

Figure 6 shows the throughput of TCP flow(s) reported by iperf for SV-NIC and HV-NIC. The setup is similar to the latency experiment described above. The mean and standard deviation for the aggregate throughput of the 'n' simultaneous flows as computed above is shown on the y-axis.

Based on these results, we make following observations:

- For a single guest VM, The performance of the *HV-NIC is about 50% of that of SV-NIC*, Several factors contribute to the performance drop for the HV-NIC, as suggested in [98], including high L2-cache misses, instruction overheads in Xen due to remapping and page transfer between driver domain and guest domains, and instruction overheads in the driver domain due to software ethernet bridging code. The overhead for software bridging is significant, as demonstrated by the difference between the dotted lines in Figure 6. In comparison, the SV-NIC adds overhead in Xen for interrupt routing and for overhead incurred in the micro-engines for layer-2 software switching.

- The performance of using a single VIF in guest VM using the SV-NIC is similar to using the tunnel interface in Dom0 without the SV-IO functionality. This shows that the cost of device-centric SV-IO realization is low and that it purely resides in the ENP2611 and the HV.

- The performance of the *HV-NIC for any number of guests is always lower than with a single VIF in the SV-NIC.*

For the last observation, there are several implementation specific issues that can explain the performance difference between a HV-NIC and a SV-NIC:

- The tunnel driver must enforce ordering over all packets and hence, it cannot take advantage of hardware parallelism effectively. In contrast, for the SV-NIC implementation, there is less contention for ordering as the number of VIFs increases. For example, on average 2 contexts will contend per VIF for ordering when #VIFs = 4, *vs.* 8 contexts, when #VIFs = 1 (or for the tunnel device in HV-NIC case). Although a smaller number of contexts per VIF implies less throughput per VIF, the aggregate throughput for all the VIFs will be more when #VIFs > 1 *vs.* #VIFs = 1 (or for the tunnel device).

- The packet receive part of the tunnel driver and the VIF device driver on the host processes packets serially, and thus can only effectively utilize a single host core. This currently impacts the HV-NIC more, since with increasing number of guest VMs,

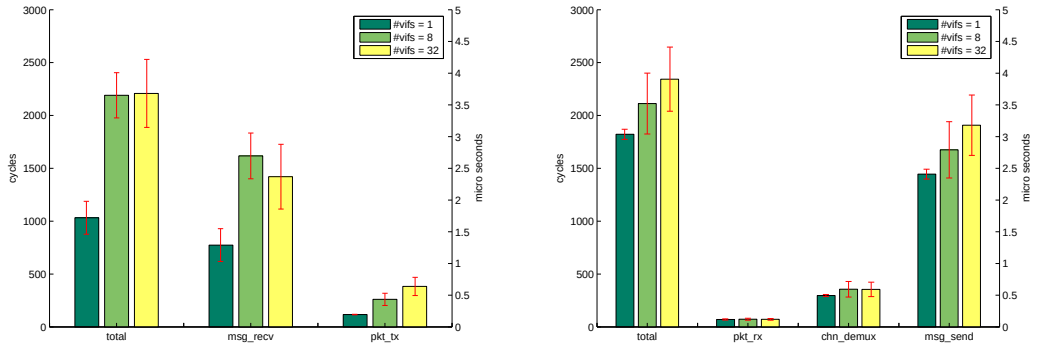
there is effectively a single core that can be used for software I/O MMU (i.e., bounce buffer copying) functionality in Dom0. With SV-NICs, since each guest VM performs its own software I/O MMU functionality, effectively two logical cores perform the copy operation (since there are two CPU cores available to run guest VMs).

- The packet send part of the tunnel driver and the VIF device driver on the NP board utilize a single micro-engine context to service the queue. With increasing number of VMs, aggregate number of contexts to service the egress path also increase for SV-NIC. However HV-NIC is currently limited, since it cannot utilize the hardware parallelism effectively.

Microbenchmark results presented later in Table 1 demonstrate that the impact of ordering of contexts by a micro-engine on NP to Host PCI write throughput is small. Hence, the latter issues of limited utilization of host and NP cores in the HV-NIC case better explains the difference in performance between a HV-NIC and aggregate SV-NICs for guest VMs ≥ 2. As discussed in Section 2.5.2, with the advent of hardware I/O MMUs, the current software I/O MMU implementation will be obviated, thereby freeing host cores from bounce buffering. This will significantly improve the receive path performance, both for HV-NIC and SV-NIC. Employing multiple micro-engine contexts to service the egress path will also improve the send path performance for both. However, we expect to see the similar level of performance difference between HV-NIC and SV-NIC for guest VMs ≥ 2 as there is in the case of a single guest VM, for the similar reasons as discussed above.

2.6.3 SV-NIC Microbenchmarks

In order to better assess the costs associated with the SV-NIC, we microbenchmark specific parts of the micro-engine and host code to determine underlying latency and throughput limitations. We use cycle counting for performance monitoring on both micro-engines and the host.



(a) Egress Path

(b) Ingress Path

Figure 7: Latency microbenchmarks for SV-NIC.

2.6.3.1 Latency

Figures 7(a) and 7(b) show the latency results for the egress and ingress paths respectively on micro-engines.

The following sub-sections of the egress path are considered:

- *msg_recv* – The time it takes for the context specific to a VIF to acquire information about a new packet queued up by the host side driver for transmission. This involves polling the send queue in SDRAM.
- *pkt_tx* – Enqueueing the packet on the transmit queue of the physical port.

For the ingress path, we consider the following sub-sections:

- *pkt_rx* – Dequeueing the packet from the receive queue of the physical port.
- *channel_demux* – Demultiplexing the packet based on its destination MAC address.
- *msg_send* – Copying the packet into host memory and interrupting the host via PCI write transactions.

The time taken by network I/O micro-engine(s) for transmitting the packet on the physical link and for receiving the packet from the physical link is not shown, as we consider it part of network latency.

When increasing the number of VIFs, the cost of the egress path increases due to increased SDRAM polling contention by micro-engine contexts for message reception from the host. The cost of the ingress path does not show any significant change, since we use hashing to map the incoming packet to correct VIF receive queue. The overall effect of these cost increases on end-to-end latency is small.

For host side performance monitoring, we count the cycles used for message send (PCI write) and receive (local memory copy) by guest domain and for interrupt virtualization (physical interrupt handler, including dispatching the signals to appropriate guest domains) by Xen via the RDTSC instruction. For $\#vifs = 1$, the host takes $\sim 9.42\mu s$ for a message receive, $\sim 14.47\mu s$ for a message send, and $\sim 1.99\mu s$ for interrupt virtualization. For $\#vifs = 8$ and 32, the average cost of interrupt virtualization increases to $\sim 3.24\mu s$ and $\sim 11.57\mu s$, respectively, while the costs for message receive and send show little variation. The cost of interrupt virtualization increases since multiple domains might need to be signaled, even redundantly in the case when $\#vifs > 8$.

2.6.3.2 Throughput

We microbenchmark the available throughput of the PCI path between the host and the NP for read (write), by reading (writing) a large buffer across the PCI bus both from the host and from the NP. In order to model the behavior of SV-NIC packet processing, the read (write) was done 1500 bytes at a time. Also, aggregate throughput is computed for 8 contexts, where all of the contexts are copying data *without* any ordering requirement among them. Results of this benchmark are presented in Figure 8.

The results show the *asymmetric nature* of the PCI interconnect, favoring writes over reads. These results validate our design choice for implementing send and receive queues in NP SDRAM and host memory, respectively.

To demonstrate the affect of ordering requirements on contexts imposed by a micro-engine in receive path processing, we perform the similar experiment as above, except this time a thread may have to wait after copying the packet in host RAM and before updating the queue data structures to ensure in-order delivery. Table 1 shows the throughput achieved

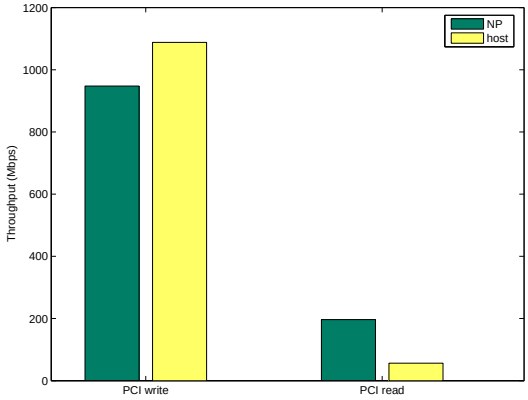


Figure 8: Throughput of the PCI interconnect between the host and the NP.

Table 1: PCI write throughput from NP to the host (Mbps).

unordered	950.694
ordered	953.679

by 8 concurrent contexts with and without the ordering requirement. We conclude from these results that ordering requirement does not impact the NP to PCI write throughput.

Our current prototype utilizes all 8 micro-engine contexts for programmed I/O in ingress path. However, increased contention among these contexts for bus access may outweigh the benefits of pipelined I/O. Experimental results depicted in Figure 9 validate this, where utilizing more than 2 contexts results in a decrease in write throughput. Hence limiting the number of micro-engine contexts may not only increase the receive throughput, rest of the contexts may be utilized to implement some additional functionality without adversely affecting the SV-NIC performance. Better ingress path performance may also be achieved via the use of DMA engines available on the NP board.

2.7 Architectural Considerations

2.7.1 Performance Impact of Virtual Interrupt Space

Currently, we only have a small (8 bit) identifier for interrupt source. Therefore, when the total number of VIFs exceeds the size of the identifier, an interrupt cannot be uniquely mapped as a signal to one VIF. This results in redundant signaling of guests domains and redundant checking for new network packets. Depending on the order in which a redundant signal is provided to domains, some domains might suffer cumulative context switching

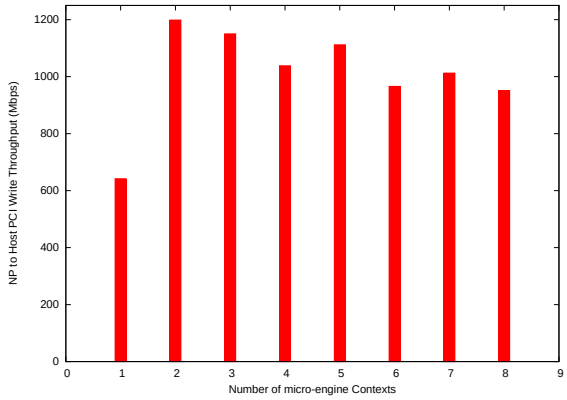


Figure 9: Effect of micro-engine contexts on NP to host write PCI throughput.

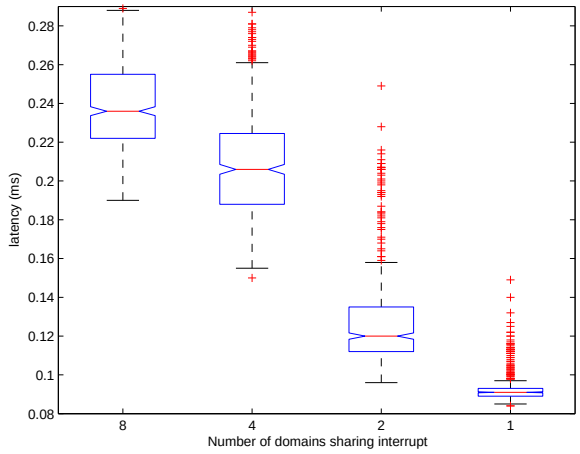


Figure 10: Effect of virtual interrupt sharing.

latency (when they cannot be scheduled simultaneously due to CPU contention). In order to demonstrate this effect, we artificially restrict the size of the identifier, ranging from 1 bit to 8 bits. This identifier space is then shared among 8 domains, each ID shared among $\lceil 8/l \rceil$ domains where l is the number of bits. We then perform a latency experiment (using the *psapp* application described above) between guests that are assigned the 'last spot' in the sharing list for an ID. This setup explores the maximum latency before a signal will be delivered to the right domain. Figure 10 shows a boxplot of the median and inter-quartile range of the latencies. As expected, latency is reduced when domains do not share interrupt IDs. *These results advocate the use of large interrupt identifiers for device-centric SV-IO realizations.*

2.7.2 Insights for Future Multi-cores

In modern computer architectures, different interconnects (buses) connect system components like memory and CPU in a particular topology. These buses are themselves connected together via bridges, thereby providing a communication path between different components.

The result of organizations like these is non-uniform communication latency and bandwidth between different components; *e.g.*, the communication path between a CPU core and memory is usually much faster and of higher bandwidth than the path between the CPU core and I/O devices. While interconnect technologies have improved both throughput and latency for I/O devices, they still situate them relatively 'far' from processing units and subject their data streams to bus contention and arbitration through the chipset path. This is particularly problematic for devices with low-latency requirements or short data transfers that cannot take advantage of bursts. In particular, *it has a negative impact on device-centric SV-IO*, which may potentially need to issue a larger number of interactions in order to signal multiple domains housing their virtual devices.

In upcoming chip multi-processor systems (CMPs), multiple CPU cores are placed closely together on the same chip [15]. Furthermore these cores may be heterogeneous [33], to include specialized cores like graphics or math co-processors and network processing engines (similar to NPs), along with general purpose cores. These cores may also share certain resources, such as L2 cache, which can greatly reduce inter-core communication latency. Our results demonstrate that *a specialized multi-core environment will better support efficient realization of device-centric SV-IO.*

To quantify and compare the architectural latency effects of the current I/O data path in the multi-core paradigm, we run a simple 'ping-pong' benchmark that passes a short 32-bit message back and forth between (1) two distinct *physical* CPU cores, and (2) a CPU core and an attached NP using shared memory *mailboxes*. For the CPU-to-NP benchmark, the local mailbox for CPU core (NP) is present in host memory (NP's SDRAM), and the remote mailbox for CPU core (NP) is present in NP's SDRAM (host memory). For the CPU-to-CPU benchmark, all mailboxes are present in host memory.

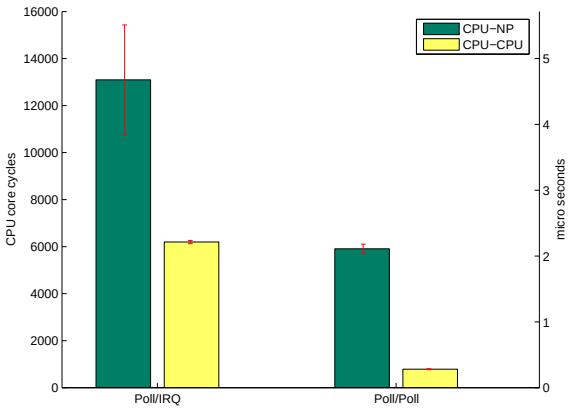


Figure 11: Comparison of communication latency for a simple ping-pong benchmark between two CPU cores, and a core and an attached NP.

We experiment with polling in both directions (Poll/Poll) *vs.* polling in one direction coupled with asynchronous notification (IRQ) in the other direction (Poll/IRQ), for receiving a message in the local mailbox from the peer. The case of using IRQs in both directions is omitted, since the host CPU cannot send such a notification to the NP's packet processing cores (micro-engines.) Inter-processor interrupts (IPI) are used to send IRQ notifications between CPU cores, while a PCI interrupt is used to send the same from NP to host CPU.

Results are reported in Figure 11. Times are for a complete round-trip measured using the RDTSC instruction on host CPU. The difference between the core-to-core results and core-to-NP results is attributable to the difference in the length and complexity of the data path messages need to traverse. Since the NP is present as a PCI device, the path between the CPU cores, the NP, and their respective remote mailboxes is 'longer' than that between two CPU cores and their memory interconnect. This extra distance adds overhead as well as variance, especially in overload conditions when the various buses' scheduling and arbitration is under stress. The difference between polling and asynchronous IRQ notifications is caused by the costs of saving and restoring the CPU context and other OS-level interrupt processing as well as demultiplexing potentially shared IRQ lines.

One tradeoff associated with spin-based polling is that it causes wasted CPU cycles, and therefore, additional energy consumption. It is possible to perform power efficient polling in recent processors via two new instructions, termed 'monitor' and 'mwait'. The CPU

programs a dedicated memory snooping circuit via the monitor instruction, providing it the target memory location to be polled. It then enters a low power sleep state via the mwait instruction. The CPU is woken up in case of a write to the target memory location, or for other reasons (*e.g.*, external interrupts). A polling loop implemented using monitor/mwait attains latency comparable to spin based polling, albeit consumes less power. Since the older Xeon host cores in our testbed do not support these instructions, we quantify the latency of this approach using a Pentium extreme edition 3.2GHz processor-based machine with attached ENP2611 board. We then scale the cycle count numbers to the original testbed cores, assuming the absolute time for ping-pong benchmark would remain the same on both cores. Results are encouraging, as they show that the benchmark takes $\sim$ 6536 cycles ($\sim 2.27\mu s$), which is similar to the case when spin-based polling is employed.

In 'true' heterogeneous multi-core processors of today and tomorrow, such as the Cell broadband engine [33], we envision a much shorter communication path between different types of cores than the one available among cores in our host-NP platform. Hence, these 'true' environments would provide much better performance for core-to-core communication. By utilizing power-efficient polling methods in such environments, we can achieve smaller core-to-core communication latency along with energy efficiency. To project these benefits in absence of a 'true' heterogeneous multi-core environment, we utilize a current generation homogeneous multi-core system and obtain core-to-core communication latency results for various types of intercore communication methods similar to the scenario above. In particular, these results are obtained on a Pentium D 3.2GHz processor-based machine. These results demonstrate that the performance of mwait based power-efficient polling is quite similar to that of spin-based polling, and is more than $3X$ better than that of interrupt based communication.

From these results, it is clear that in the approaching multi-core world, there will be substantial benefits to be attained by (1) *positioning NP-like communication cores 'close' to the host computer cores*, and (2) *having many rather than few cores* so they can be dedicated to particular tasks and thus allow use of low-latency polling rather than slow and variable asynchronous interrupts. The communication latencies for polling-based solutions will be

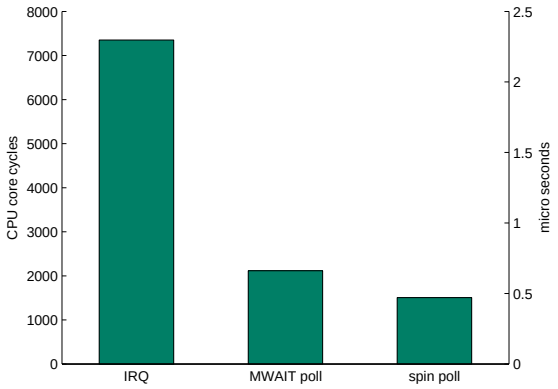


Figure 12: Communication latency for a simple ping-pong benchmark between two CPU cores.

further reduced when these cores share resources like L2 cache, as that would reduce/eliminate the costs associated with cache invalidations and accessing system memory. The cost of interrupt-based solutions may also be reduced if the system bus connecting the interrupt controllers of all cores (*e.g.* LAPICs for x86 architecture) is also implemented on chip.

2.7.2.1 In-Place Data Manipulation

Specialized cores typically provide functionality that is costlier to implement on general purpose processors. For example, network processors perform network-specific processing on data, which can also include application-specific data processing, such as filtering information from message streams based on certain business rules [70]. In a multi-core system, the resultant data/information must also be made available to other cores in case they are executing other parts of application logic. Although there is resource sharing in modern systems to enable this (*e.g.*, the host CPU can access NP's SDRAM and vice-versa), *often the cost of accessing shared resources makes in-place data manipulation by different cores prohibitively costly* (as shown by our microbenchmarks, both NP and host PCI read bandwidth are very limited). As a result, multiple data copies must be made by different cores to their *local* memory before they can efficiently operate on it. This, coupled with the fact that application logic running at one core may not have complete knowledge of the information requirements of other cores, may result in large amounts of wasted memory bandwidth and increased latency due to redundant data copying.

Future multi-core systems will alleviate this problem, since all cores will be equidistant from main memory and hence will be able to access shared information at similar cost. However, the cost of accessing memory may still become a bottleneck in the case of collaborative in-place data manipulation by multiple cores. Early trends demonstrate that future multi-core systems will share caches at some level (*e.g.*, L2 cache) [15], but the increasing number of cores will raise multiple issues with cache sharing. First, in a multiple-CMP configuration where each chip has only a small number of cores, coherency will be an issue among caches on different chips, and will require more complex cache coherence protocols, such as token coherence [94]. Second, for large-scale CMPs, with many cores on the same chip, large shared caches will no longer have a uniform access time, rather, that time will depend on the wire distance between the core and the specific part of the cache being accessed [80]. This might require restructuring applications' access behavior in order to extract good overall performance.

2.8 Conclusions and Future Work

In this chapter, we advocate the SV-IO abstraction for I/O virtualization. We also enumerate various design choices that can be considered for the realization of this abstraction on modern computer systems. Specifically, we present the design and an initial implementation of a device-centric SV-IO realization as a *self-virtualized network interface device* (SV-NIC) using an IXP2400 network processor-based board. Performance of the virtual interfaces provided by this realization is analyzed and compared to a host-centric SV-IO realization on platforms using the Xen hypervisor. The performance of device-centric SV-IO is better than that of the host-centric SV-IO. It also scales better with an increasing number of virtual interfaces used by an increasing number of guest domains.

Our SV-NIC enables high performance in part because of its ability to reduce HV involvement in device I/O. In our solution, the HV on the host is responsible for managing the virtual interfaces presented by the SV-IO, but once a virtual interface has been configured, most actions necessary for network I/O are carried out without HV involvement. Here, a limiting factor of our current hardware is that the HV remains responsible for routing

the interrupt(s) generated by the SV-IO to appropriate guest domains. Future hardware enhancements, such as larger interrupt ID spaces and support for message signaled interrupts may alleviate this problem. while hardware support for allowing interrupts to be routed directly to guest domains [17] may relieve the HV of interrupt routing responsibility altogether.

There are certain optimizations that can improve the performance of VIFs for both HV-NIC and SV-NIC implementations. Specifically, we can:

- Improve upstream throughput by replacing micro-engine programmed I/O with DMA. Further, by utilizing a NP-platform with near-identical PCI read and write performance [23], we can also replace programmed I/O performed by host cores with DMA. This will reduce the host CPU utilization.

- Improve TCP performance via TCP segment offload. This will reduce host CPU utilization.

- Add support for large MTU sizes (jumbo frames). This will provide better utilization of resources and can improve performance by replacing multiple small bounce buffers with fewer large bounce buffers.

For host-centric HV-NIC, the driver domain can be specialized into a lean 'stub-domain'. The goal is to implant only the functionality required for this purpose, rather than using a full fledged driver domain. A stub-domain also allows for methods that reduce the impacts of OS noise [107]. For example, by utilizing 'smart' polling-based I/O and tickless approach for time keeping [38], the network stub-domain can minimize the impact of interrupt processing on virtualized network I/O performance in an energy efficient manner. This network stub-domain can be further enhanced to perform the device emulation required to virtualize I/O for un-modified guest VMs (HVM domains) in kernel, as compared to the current approach where device emulation is performed in userspace in Dom0. This will increase the performance of network I/O by removing numerous kernel-user boundary crossings that currently take place in Dom0.

CHAPTER III

RE-ARCHITECTING VMMS FOR MULTICORE SYSTEMS: THE *SIDECORE* APPROACH

Future many-core platforms present scalability challenges to VMMs, including the need to efficiently utilize their processor and cache resources. Focusing on platform virtualization, we address these challenges by devising a virtualization method that utilizes the fact that cores will differ with respect to their current internal processor and memory states. The hypervisor, or VMM, then leverages these differences to substantially improve VMM performance and better utilize these cores. The key idea underlying this work is simple: to carry out some privileged VMM operation, rather than forcing a core to undergo an expensive internal state change via traps, such as VMexit in Intel's VT architecture, why not have the operation carried out by a remote core that is already in the appropriate state? Termed the *sidecore* approach to running VMM-level functionality, it can be used to run VMM services more efficiently on remote cores that are already in VMM state. This work demonstrates the viability and utility of the sidecore approach for two VMM-level classes of functionality: (1) interrupt virtualization for self-virtualized devices. and (2) efficient VM-VMM communication in VT-enabled processors.

3.1 Sidecores: Structuring Hypervisors for Many-Core Platforms

Current VMM designs are monolithic, that is, all cores on a virtualized multi-core platform execute the same set of VMM functionality. We advocate an alternative design choice, which is to structure a VMM as multiple components, with each component responsible for certain VMM functionality and internally structured to best meet its obligations. As a result, in multi- and many-core systems, these components can even execute on cores other than those on which their functions are called. Furthermore, it becomes possible to 'specialize' cores, permitting them to efficiently execute certain subsets of rather than complete sets of VMM functionality. A similar componentization approach in multi-processor systems is taken by

the K42 operating system [84].

There are multiple reasons why functionally specialized, componentized VMMs are superior to the current monolithic implementations of VMMs, particularly for future many-core platforms:

1. Since only specific VMM code pieces run on particular cores, performance for these code pieces may improve from reductions in cache misses, including the trace-cache, D-cache, and TLB due to reduced sharing of these resources with other VMM code. Further, assuming VMM and guest VMs do not share a lot of data, VMM code and data are less likely to pollute a guest VM's cache state, thereby improving performance isolation for guests and improving overall guest performance.

2. By using a single core or a small set of cores for certain VMM functionality (e.g., page table management), locking requirements may be reduced for shared data structures, such as guest VM page tables. This can positively impact the scalability of SMP guest VMs.

3. When a core executes a VMM function, it is already in the appropriate processor state for running another such function, thus reducing or removing the need for expensive processor state changes (e.g., the VMexit trap in Intel's VT architecture).

4. In heterogeneous multicore systems, some of these cores may be specialized and hence, can offer improved performance for doing certain tasks compared to other non-specialized cores [87].

5. Dedicating a core can provide better performance and scalability for the I/O virtualization path, as demonstrated in Section 3.2.

6. To take full advantage of many computational cores, future architectures will likely offer fast core-to-core communication infrastructures [118], rather than relying on relatively slow memory-based communications. The sidecore approach can leverage those technology developments. Initial evidence is high performance intercore interconnects, such as AMD's HyperTransport [10] and Intel's planned CSI.

In this work, we propose sidecores as a means for structuring future VMMs in many-core systems. The current implementation dedicates a single core, termed *sidecore*, to perform specific VMM functions. This sidecore differs from *normal* cores in that it only executes one or a small set of VMM functionality, whereas normal cores execute generic guest VM and VMM code. A service request to any such sidecore is termed a *sidecall*, and such calls can be made from a guest VM or from a platform component, such as an I/O device. The result is a VMM that attains improved performance by internally using the client-server paradigm, where the VMM (server) executing on a different core performs a service requested by VMs or peripherals (clients). We demonstrate the viability and advantages of the sidecore approach in two ways. First, the sidecore approach is used to enhance the I/O virtualization capabilities of *self-virtualized devices* via efficient interrupt virtualization. Second, we briefly describe how a sidecore is used to perform efficient routing of service requests from the guest VM to a VMM, to avoid costly *VMexits* in VT-enabled processors. The latter use case is described in details elsewhere [88].

3.2 Enhancing Interrupt Virtualization for Self-virtualized Devices

In current virtualized systems, e.g., those based on the Xen VMM, I/O virtualization is typically performed by a driver domain, which is a privileged VM with direct access to the physical device. For 'smart' devices with on-board processing capability, an alternative is to offload parts of the I/O virtualization functionality from the driver domain onto the device itself. These devices, hereafter termed self-virtualized devices, provide a direct, low-latency I/O path between the guest VM and physical device with minimal VMM involvement. This model of VMM bypass I/O improves performance and scalability [111, 91]. We have implemented a self-virtualized network interface (SV-NIC) using an IXP2400 network processor-based gigabit ethernet board [6], as described earlier in Section 2.4. This SV-NIC provides virtual network devices, hereafter termed as VIFs, to guest VMs for network I/O. A guest VM enqueues packets on the VIF's send-queue and dequeues packets from the VIF's receive-queue.

In the egress path, the micro-engines, which are part of the IXP2400 NP, poll for packets

in the guest VM's send-queue, which obviates the need of any involvement of the VMM or the driver domain. However, in the ingress path, the SV-NIC needs to signal the guest VM when packets are available for processing on the receive queue. Since the SV-NIC is a PCI device, it does so by generating a single master PCI interrupt, which is then routed to a host core by the I/O APIC. The master interrupt is intercepted by Xen, and based on the association between bits in the identifier register, VIFs and guest VMs, a signal is routed to the appropriate guest VM(s). Specifically, the master PCI interrupt is generated by the SV-NIC via a 8-bit wide identifier register – setting any bit in this register generates the master interrupt on the host. Hence, the SV-NIC can uniquely signal guest VMs for up to 8 VIFs. However, when the number of VIFs exceeds 8, these bits are shared by multiple VIFs, which may result in redundant signaling of guest VMs and may cause performance degradation.

There are multiple reasons for this design: (1) an ample number of micro-engines and fast switchable hardware contexts make it cheaper to poll for information than to wait for an asynchronous signaling mechanism like an interrupt; (2) hardware contexts running on micro-engines are non-preemptible, thus the context must explicitly check for the presence of interrupt signal anyway; and (3) there exists no direct signaling path from host cores to micro-engines, hence if signals were to be used for egress path, such signals must be routed via the XScale core, resulting in prohibitive latency.

In the sidecore approach, we use a host core to carry out the interrupt virtualization task. We establish a separate messaging channel between the micro-engines and the sidecore. This messaging channel is created in host-memory accessible via PCI I/O to micro-engines and local memory I/O to the sidecore. In ingress path, micro-engines enqueue a message containing the ID of the VM that requires signaling. The sidecore continuously polls this messaging channel, and based on the ID contained in the message, sends a signal to the corresponding VM, signifying that one or more packets are available in the receive queue of the VNIC. As described in Section 2.7.2, there are alternatives to the sidecore's polling approach that may result in better energy savings, while providing similar level of latency benefits.

This approach not only improves performance, by avoiding the need for explicit interrupt routing, but it also improves performance isolation between multiple guest VMs. One example is a signal sent by SV-NIC for a VIF whose corresponding guest VM is not currently scheduled to run on the host core where the master interrupt is delivered. In our current implementation without sidecore, such a signal unnecessarily preempts the currently running guest VM to the 'Xen context' for interrupt servicing. In contrast, the sidecore approach uses one host core exclusively for interrupt virtualization and hence, does not interrupt any guest VM unnecessarily. Further, we abandon signaling via PCI interrupts altogether which reduces the latency of the signaling path by avoiding redundant signaling. A negative element of the approach is that all signals must always be forwarded by the sidecore to the core running the guest domain as an inter-processor interrupt (IPI). That is, in this case, it is not possible to opportunistically make an upcall from the host core processing master interrupt to the guest domain in case the intended guest domain is currently scheduled to run on the same host core.

3.2.1 Evaluation

In this section, we compare the latency of the network I/O path for guest VMs using SV-NIC provided VIFs to demonstrate the performance and scalability benefits of the sidecore approach. The experiment is conducted across two host machines. Each host is a dual core, two-way hyperthreaded, Pentium Xeon (a total of 4 logical processors) 2.80GHz server, with 2GB RAM. The VMM used for system virtualization is Xen 3.0-unstable. Each VM runs a paravirtualized Linux 2.6.16 kernel with a small ramdisk root filesystem based on the Busybox distribution and is configured with 32MB RAM. For the SV-NIC implementation enhanced with sidecore functionality, logical processor 0 is assigned to Dom0, while logical processor 1 is used as the sidecore (both of these belong to the same CPU core). For the SV-NIC without sidecore, logical processors 0 and 1 are both assigned to Dom0. The other two logical processors (belonging to the same CPU core) run the guest VMs. These logical processors share many architectural resources among them, such as caches and execution units. However, each logical processor has its own local APIC [93]. Hence, an interrupt can

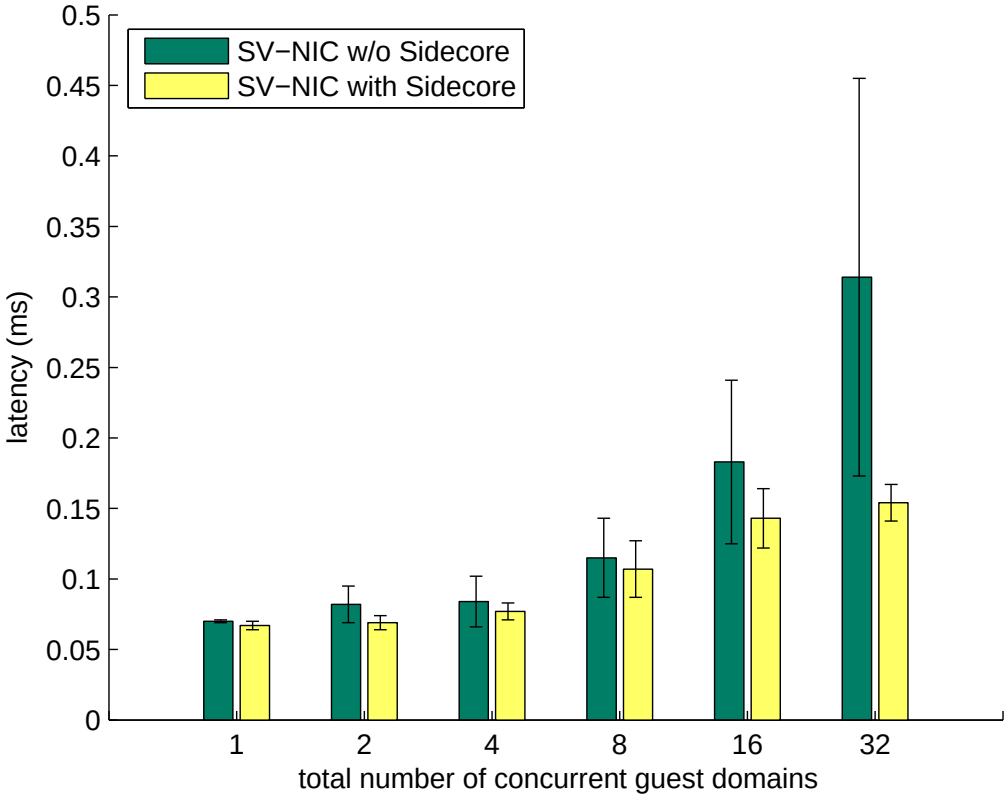


Figure 13: Latency of network I/O virtualization for SV-NIC without any sidecore and SV-NIC with a sidecore.

be directed to a specific logical processor, without impeding the other one sharing resources with the target logical processor. The Borrowed Virtual Time (bvt) scheduler with default arguments is the domain scheduling policy used for Xen.

For latency measurements, the experimental setup is similar to that in Section 2.6. Figure 13 compares the RTT reported by *psapp* comparing the SV-NIC without the sidecore and the SV-NIC with a sidecore for interrupt virtualization, as the number of guest VMs is increased on both hosts.

Results demonstrate that there is not much benefit from using the sidecore approach for up to 8 guest VMs, since a signal can be uniquely delivered to a certain guest VM. Beyond 8 guest VMs, the SV-NIC without sidecore causes redundant signaling, which adversely impacts latency. For example, for 16 guest VMs, there are .5 redundant guest VMs being scheduled for network I/O per packet on average. In comparison, there is no redundant

signaling for the SV-NIC with sidecore, thereby improving latency. Larger latency gains are obtained for 32 guest VMs because of yet more redundant signaling.

In terms of variability as depicted by the inter-quartile range, the sidecore approach provides less variability since the cost of signaling for every VM is similar - the sidecore signals the core where the target VM is executing via an IPI. In the case without the sidecore, the opportunistic signal delivery as described earlier where a target VM for the signal may be scheduled to run on the same core which is interrupted results in more variability. Redundant signaling further exacerbates the variability.

3.3 Efficient Guest VM-VMM Communication in VT-enabled Processors

Earlier implementations of the x86 architecture were not conducive to *classical* trap-and-emulate virtualization [44] due to the behavior of certain instructions. System virtualization techniques for x86 architecture included either *non-intrusive* but costly binary rewriting [36] or efficient but highly intrusive paravirtualization [48]. These issues are addressed by architecture enhancements added by Intel [17] and AMD [4]. In Intel's case, the basic mechanisms in VT-enabled processors for virtualization are *VMentry* and *VMexit*. When the guest VM performs a *privileged* operation it is not permitted to perform, or when guest VM explicitly requests service from the VMM, it generates a VMexit and the control is transferred to the VMM. The VMM performs the requested operation on guest's behalf and returns to guest VM using VMentry. Hence, the cost of VMentry and VMexit is an important factor in the performance of implementation methods for system virtualization.

Microbenchmark results presented in Figure 14 compare the cost of VMentry and VMexit with the intercore communication latency experienced by the sidecore approach. These results are gathered on a 3.0 GHz dual-core X86-64 bit, VT-enabled system, running a uni-processor VT-enabled guest VM (hereafter referred to as hvm domain). The hvm domain runs an unmodified Linux 2.6.16.13 kernel and is allocated 256MB RAM. The latest unstable version of Xen 3.0 is used as the VMM. The figure shows the VMexit latency for three cases when the hvm domain needs to communicate with the VMM: (1) for making a 'Null' call where VMCALL instruction is used to cause VMexit but VMM immediately

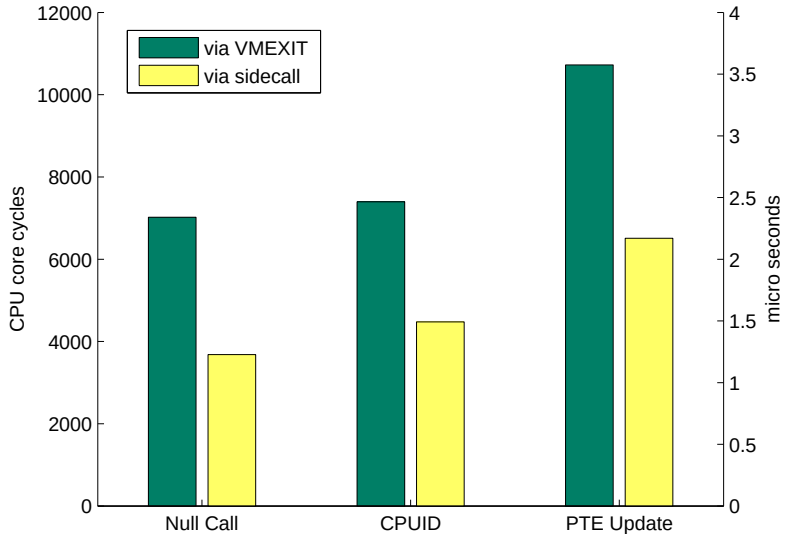


Figure 14: Latency comparison of VMexit and sidecall approach.

returns; and (2) for obtaining the result of CPUID instruction which causes VMexit and VMM executes the real CPUID instruction on hvm domain's behalf and returns the result.

The figure also presents comparative results when VM-VMM communication is implemented as a sidecall using shared memory channel. In particular, one core is assigned as the sidecore, and the other core runs the hvm domain, with a slightly modified Linux kernel. When the hvm domain boots, it establishes a shared page with the sidecore to be used as a communication channel. The operations mentioned above are implemented as synchronous shared memory requests to avoid VMexits. In the first case ('Null' call), the sidecore immediately returns a success code via the shared memory. In the second case, it executes the CPUID instruction on the hvm domain's behalf and returns the result.

Results demonstrate considerably higher performance for the sidecall compared to the VMexit path. Hence, by replacing VMexits with cheaper sidecore calls, the performance of the hvm domain and of system virtualization overall can be improved significantly. To implement the sidecalls described above, only 7 lines of code was modified in the guest kernel and only $\sim$ 120 lines of code in the form of a new kernel module was added.

Besides these microbenchmarks, sidecalls can also benefit a guest VM's page table management by reducing the overall number of VMexits, and hence reducing latency, in guest's

page fault handling. These results, presented in detail elsewhere [88], show the approach's benefits, on average providing 41% improvements in latency.

3.4 The Sidecore Approach: Discussion

The performance benefits of the sidecore approach for VM-VMM communication might become less pronounced as VMexit/VMentry operations are further optimized [44]. However, we believe that the sidecore approach can still provide significant advantages. First, since low latency inter-core interconnects are important for attaining high performance for parallel programs on future many-core platforms, they are likely to be an important element of future hardware developments. The latency of intercore communication can be further decreased by cache sharing among cores or by direct addressed caches from I/O devices [148]. Second, re-architecting the virtualized system as virtualization services via sidecores provides a clear separation of functionality between VM and VMMs which might imply better performance for VMs due to reduced VMM *noise*, caused by the pollution of architectural state. Moreover, it has been shown that using functional partitioning is one of the important techniques for improving scalability of system software in large scale many-core and in multiprocessor systems [118, 67].

One disadvantage of the sidecore approach is that its current implementation requires minor modifications to the guest OS kernel. However, these changes are significantly smaller than a typical paravirtualization effort – $\sim$ 120 lines for setting up the shared communication ring and 7 lines for sending a sidecall request. Hence, the approach has a desirable property of *minimal* paravirtualization. Besides, the approach can be dynamically turned on/off with a simple flag, allowing the same guest kernel binary to execute on a VMM with-/without the sidecore design. Another trade-off is that the sidecore approach causes wasted cycles and energy due to the CPU spinning used to look for requests from guest VMs. This can be alleviated via energy-efficient polling methods, such as the monitor/mwait instructions available in recent processors. A final issue is that the use of sidecores to run specialized functions might make them unavailable for normal processing. A sidecore implementation that dynamically finds available cores would alleviate this problem, but we

have not yet implemented that generalization and therefore, cannot assess the performance impacts of runtime core selection. For a static approach, we hypothesize that in future large-scale many-core systems like those in Intel's tera-scale computing initiative [32], it will be reasonable to use a few additional cores on a chip for purposes like these, without unduly affecting the platform's normal processing capabilities.

3.5 Conclusions and Future Work

This chapter presents the sidecore approach to enhance system-level virtualization in future multi- and many-core systems. The approach factors out some parts of the VMM functionality in order to execute it on a specific host core, termed *sidecore*. We demonstrate the benefits of this approach by using it to avoid costly VMexits on VT-enabled processors and by using it to improve the performance of self-virtualized devices. Performance results demonstrate that the sidecore approach improves the overall performance of the virtualized system.

Future work will address policies and mechanisms to dynamically deploy sidecore functionality on a normal core. Specifically, the VMM can periodically gather the CPU utilization from the scheduler to determine which cores are a good target to execute dedicated sidecore functionality for a specific period of time. In case the processing resources are underutilized, this dynamic approach provides the performance and isolation benefits of sidecore for currently active VMs.

CHAPTER IV

ENABLING SEMANTIC COMMUNICATIONS FOR VIRTUAL MACHINES VIA LOGICAL DEVICES

In virtualized environments, optimized virtual machine communication is highly important to overall system performance. Focusing on the communication related to I/O performed by VMs, and leveraging the fact that modern systems already have to virtualize the physical devices used by VMs, this chapter shows the benefits of extending existing virtual device interfaces for building virtualization services. Specifically, by devising *enhanced* virtual devices, termed *logical devices*, we can:

- efficiently implement the communication paths between virtual machines (VMs) and the virtualized platforms (VPs) on which they run, and
- capture semantic information about VM-device interactions, which can then be used to implement additional functionality and efficient sharing of physical devices.

The chapter presents two concrete examples of virtualization services that provide logical devices: a network virtualization service that provides virtual NICs with QoS-support where the VM communicates its QoS requirements to the VP, and a storage virtualization service which permits a VM to access a block device regardless of whether such a device is physically located locally or must be accessed at a remote location. Xen-based implementations of these services demonstrate substantial performance improvements and additional functionality derived from the corresponding logical devices at a minimal cost to VMs, in part because virtualization services can utilize additional computational resources of the VP and can take better advantage of certain underlying platform capabilities.

4.1 Introduction

The thesis of this chapter is that virtualized platforms (VPs) should go beyond basic virtualization services – those offering virtualized execution environment with high performance

communication, to also implementing efficient *end-to-end services* across different virtual machines and between VMs and external application end-points or data sources. Examples of such services include isolation guarantees across multiple VMs' interactions and different levels of quality of service for VMs that vary in speeds or capabilities. Implementations of these enhanced virtualized services may reside in driver domains running on the general purpose host processor(s) in the platform, or they may be partially supported by device-resident functionality. For instance, Infiniband's network adapters [91] and the self-virtualized NIC (SV-NIC) developed in our own work [111] execute driver domain functionality that enables high performance communication and provide performance isolation.

In this chapter, we propose to take another step in the direction of improved functionality for VMs, by providing VMs with a view of an interconnect that provides meaningful information, rather than just raw data. We term the abstraction describing this semantically richer interconnect InfoConnect, or *iConnect*. Given basic VM technology, a key building block for realizing iConnect is a VM's interface to the VP. In fact, a slight enhancement of this interface makes it possible to extend/modify the raw data movement functionality supported by current VPs to provide the semantically richer *information* required by a VM. Using this extended interface, it is possible to construct information-centric iConnects that provide meaningful information to VMs in the forms in which they need it. A simple example is an iConnect addressing byte ordering mismatches between the VMs running on machine connected via some physical interconnect. This issue can be addressed by a new or reconfigured host- or device-resident driver domain component, which exports to the VM the data formats needed to deal with differences in endian-ness and which implements the data transformations that correct for mismatches.

iConnect takes an information-centric view of how communication happens in the end systems, where virtual machines executing on different CPU cores (and the applications they run) use the data sent and received for certain tasks and when doing so, extract or derive semantically meaningful information from such data. Given this view, past work has already demonstrated many useful methods for accelerating such derivations, by providing semantic information at lower levels, e.g., active disks [123] and enriched network

interfaces [70]. Leveraging these methods, iConnect's approach is to permit end systems to associate with VMs' data communications the information extraction, annotation, and/or data conversion tasks they wish to have performed on said data. The goal of such associations is to improve certain end system characteristics, such as performance, scalability, and reliability, and to provide VMs with additional functionality at no or minimal cost. Performance improvements are derived from dealing with impediments in the critical paths of information exchanges. The aforementioned data conversions to correct for differences in endian-ness constitute one such impediment. Others include dealing with how I/O is handled in virtualized systems, e.g., by OS/VMM bypass, or how I/O resources are allocated among VMs to meet their QoS requirements. Examples of services related to scalability and reliability include online monitoring to help a virtual machine better manage a platform's resources, such as memory [79], and online monitoring to isolate misbehaving VMs.

The iConnect abstraction includes the mechanisms that permit the VP to extract semantically meaningful information about the VM. In this chapter, our focus is on a VM's I/O, where the VP associates some computation with a device I/O path (on the host-based driver domain, or on the physical device). An iConnect realized in this manner provides the VM with an enhanced virtual device, termed a *logical device*, with additional attributes/-functionality which may not be natively supported by the corresponding physical device.

Before continuing, we note that it is difficult to cleanly distinguish 'data' from 'information'. This is because the same unit may be treated as data by some software modules and as information by others. A concrete example is a unit of block device data handled by the device driver layer. It may contain file system specific directory information, i.e., information from the point of view of the file system, or data in a memory page from the operating system's point, or a structure implementing a binary search tree from an application's point of view. iConnect, therefore, leaves it up to the end systems that handle these units to state and implement their information-centric views.

There are multiple technical motivations for developing iConnect. One is to mitigate the increasing mismatch in CPU vs. memory speeds in modern processor architectures. The iConnect approach does so by using additional processing cycles to implement semantically

meaningful data manipulations for communicating virtual machines. Another motivation is to make efficient use of the increased processing power and concurrent operation offered by modern multi-threaded and multicore processors. The iConnect abstraction addresses this issue by permitting developers to flexibly exploit both device- and host-resident processing resources. Finally, by shifting computations related to I/O to the VP, the iConnect approach essentially provides to guest VMs additional functionality with minimal or no computational cost to them. This can permit them to function better in the presence of resource restrictions, such as limited availability of cores to VMs or licensing restrictions imposed by software when VMs use certain numbers of cores.

In the remainder of this chapter, we first describe the iConnect abstraction. This is followed by concrete iConnect realizations focused on the I/O interactions of VMs. These realizations provide the following logical devices to VMs:

- a QoS enhanced virtual network interface, and
- a remote virtual block device (RVBD).

Experimental results demonstrate that these realizations provide desired logical functionality at a considerably lower cost than prior OS-level solutions.

4.2 iConnect

The key element of the iConnect abstraction is the VM-VP communication interface, through which the original VP is extended to provide semantically richer data exchanges. In modern virtualized platforms, this interface is implemented in one of the following ways:

- *Hypercall and upcall* – a VM's service requests to the VMM via the hypercall interface use a shared memory area and/or register state to pass parameters. Similarly, the VMM can provide notification to a VM via an upcall; it can also send a signal, a.k.a. a *virtual interrupt*, to the VM. This also includes *sidecall*, a specific form of hypercall as defined in Chapter 3.
- *Shared memory message channel* – VM service requests to the Service VM are sent via a shared memory channel that carries iConnect-specific semantic information.

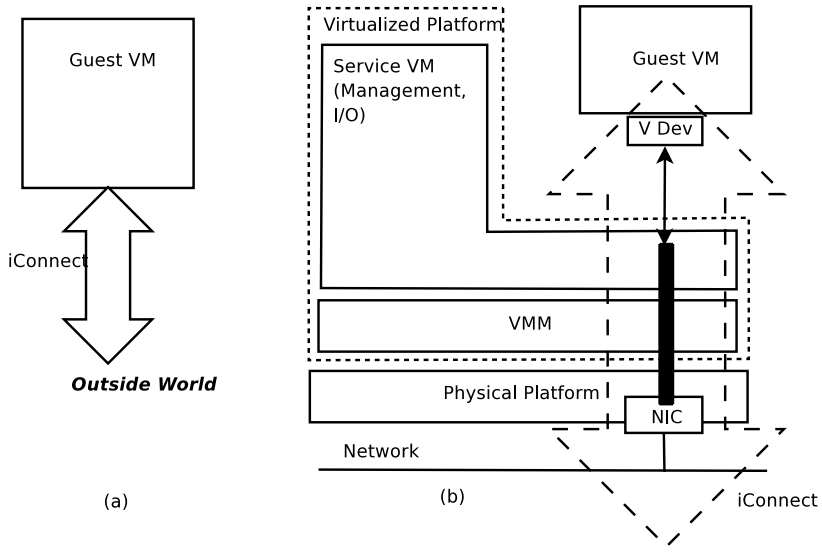


Figure 15: The iConnect abstraction (a) high level view (b) basic concept, dark box shows the possible entities where iConnect functionality may be implemented.

Inter-VM signaling is implemented via hypercalls through the VMM.

- *Safe direct access* – for devices with VMM-bypass capabilities, where (part of) the driver functionality is executed on the device itself, the interface is implemented via an I/O bus and shared memory, and it permits direct device access to VM memory, and vice versa, along with accompanying signaling and coordination mechanisms.

In all of these cases, the VM's interaction with virtual devices for *data exchange* is via the original API supported by the device.

The second element of the iConnect abstraction is comprised of a set of mechanisms used by the VP to gather information about the VM's requirements for semantic information. In this chapter, our focus is on a VM's I/O requirements, which are met by having the VP associate some additional computation with a device's I/O path. These computations are run on the host-based driver domain, on the physical device, or split across both. The corresponding implementation of this functionality results in what are best described as enhanced virtual devices, which we term *logical devices*. A guest VM using such a device will see additional attribute(s) and/or new functionality that may not be natively supported by the corresponding physical device. The computational resources used by the VP for implementing logical device functionality are separate from those used by the guest VM.

Examples of such resources are additional host-side cores or computational facilities resident on the network processor 'close to' the physical network interface.

The API provided to the VM(s) for the information transported via the iConnect is dependent upon the kind of data path that is being enriched by the iConnect. Typically, additional semantically enriched functionality is a superset of the basic 'raw' data transport functions supported by the physical device that is being extended. For instance, in earlier work, we implemented a logical camera device that focuses a camera's output on some concrete region and then hides certain elements of the recorded image based on end-user credentials (e.g., blurring people's faces) [82]. When used with iConnect, these logical camera devices provide VMs with an API to specify region coordinates and include authentication information, along with the original API used to gather image data. Regardless of these details, any such API is realized by the mechanisms described earlier that implement the VM-VP communication interface.

4.2.1 Implementation Detail

The software framework supporting the iConnect abstraction is realized for virtualized platforms built with the Xen VMM [110]. The basic Xen VMM virtualizes core architectural resources, such as CPU cores and memory, while I/O resources are virtualized using driver domains. In the driver domain approach, the iConnect abstraction uses shared memory communication between VM and VP. Additional functionalities and properties required to implement logical devices are implemented inside the driver domain. Alternatively, using 'smart', self-virtualized devices, the functions run in driver domains are instead executed by the device itself [111]. The concrete example of such a *self-virtualized device* used in this chapter is a self-virtualized NIC, which in our implementation, consists of an IXP2400 network processor-based board with a gigabit ethernet port, connected to the host system as a PCI device.

For the former driver-domain based approach, the overall I/O path (and hence the latency) experienced by data to proceed from the physical device to the VM is longer, since the VP must schedule and run multiple VMs for each interaction. However, the cost of

implementation is low due to the ease of programmability on standard x86 hardware. For the latter self-virtualized device based approach, the latency experienced by data movement is reduced, since data moves directly from device to guest VM, as described in Section 2.4. However, with more functionality desired from iConnect realizations, the cost of building a solution with this approach increases due to the increased complexity and cost of software development on a specialized platform. In this chapter, the example iConnect realizations belong to the latter, self-virtualized device approach. Examples of iConnect realizations based on driver-domain approach are presented in Chapter 5 and 6, where we describe two services, multimedia virtualization and object-based storage virtualization, respectively, in detail.

4.2.2 QoS Enhanced Self-Virtualized NIC

Dynamic VM/application behaviors and consequent changes in the resource needs of their data flows require that the iConnect be aware of VMs' quality requirements. Furthermore, VMs must communicate these requirements to the VP.

For network I/O, we enhance the self-virtualized NIC (SV-NIC) prototype described in [111] with priority-based QoS support, where flows from different VMs can be assigned different priorities. The prototype is based on the aforementioned IXP2400 network processor based board containing a gigabit ethernet interface. The NIC provides virtual interfaces directly to the guest VMs, with minimal VMM interaction in the network I/O path over iConnect. A device driver for the virtual NIC interface forms the VM-side endpoint of iConnect. The device driver provides an IOCTL-based interface to the guest VM for communicating QoS requirements, i.e., a numeric priority value. Next, we modify the iConnect's path responsible for SV-NIC management to incorporate the communication of the QoS attribute, which is implemented as a VM-VMM *hypercall*. This hypercall is implicitly called by the driver domain as a result of aforementioned IOCTL, thereby resulting in information exchange over iConnect.

The VP, specifically the SV-NIC, uses the information sent by guest VMs to compute scheduling policies and resource allocation requirements for all VNICs corresponding to

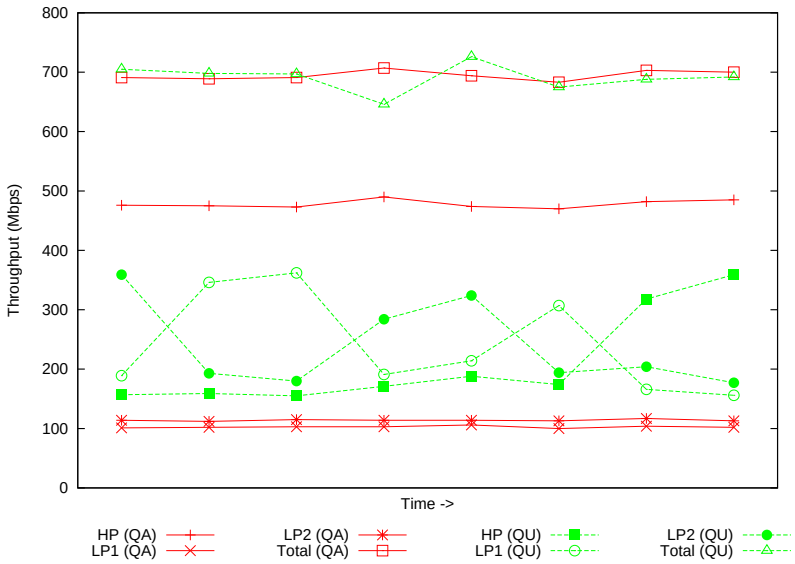


Figure 16: Quality-Aware (QA) vs. Quality-Unaware (QU) iConnect.

all VMs, where resources include network processor resources, such as IXP microengine contexts, and memory resources available on the SV-NIC. A more detailed description of how the QoS feature is implemented appears elsewhere [100].

Figure 16 shows the benefits derived from the QoS-enhanced iConnect. In this experiment, three VMs, with one VNIC each, are used to perform network I/O. One VM is set at high priority (HP), and the other two are set at lower priority (LP1 and LP2). The priority information is communicated via the iConnect, as described earlier. The quality-aware (QA) iConnect correctly recognizes a situation of high input rate for all three flows and switches to providing guaranteed throughput to the indicated flow. In contrast, the quality-unaware (QU) iConnect lacks the functionality to discriminate across different QoS requirements, which results in all flows receiving a random percentage of the total egress throughput at any point in time.

4.2.3 Remote Virtual Block Device

A simple logical functionality currently implemented by guest operating systems is that of a network block device (NBD) [22]. With NBD, block devices (e.g., disks) can be accessed remotely by having the guest operating system extract disk block information from the network packets it receives. iConnect enables an alternative approach that provides to

guest VMs transparent remote device accesses. In this approach, an iConnect-realized remote virtual block device (RVBD) hides from the VM the fact that the physical block device it uses (e.g., a disk) is located remotely. Particularly, the iConnect carries disk block information, which is extracted by the VP from the network data.

The potential advantages of this approach are multi-fold. First, 'lean' functionality like that of RVBD can be implemented by the hardware, in a manner similar to iSCSI [1]. This is not likely the case for file system based realizations of remote data accesses. Second, there is already ongoing work that aims to decouple the efficient remote data accesses realized by approaches like RVBD from the complex semantics of modern file systems. An example is the Light-Weight File System (LWFS) created for the high performance domain [101] which separates fast path file read and write operations from operations used for meta-data purposes, such as file naming or consistency. The iConnect approach makes it easy to vary the placement of different elements of LWFS and/or its backend storage functionality (e.g., object stores [72]) into and/or outside the virtualized platforms being used for their implementation. Third, decoupling the device access from device location significantly helps in device consolidation in large computing systems. Fourth, this approach removes the requirement that the guest VM runs the networking stack for disk access, thereby reducing the guest's computational resource needs. Finally, having transparent access facilitates *virtual device migration* [86] while doing VM migration, i.e., it provides continued access to I/O devices during and after the guest VM migration.

Before describing the RVBD implementation with iConnect, we briefly outline the Netbus mechanism [86], an extension of the basic Xenbus mechanism used for virtual device access in systems virtualized with Xen [110]. This mechanism utilizes Xen's 'split' implementation of device driver stacks. Here, each split stack consists of a frontend (FE) and a backend (BE) driver. The VM executes the FE, and the VP executes the BE (either in the driver domain or in the self-virtualized device). FE-BE (aka, VM-VP) communicate with each other over iConnect and to enable these communications to extend across multiple machines, Netbus extends Xen's existing single-platform solution by further splitting the BE into two components, local BE (LBE) and remote BE (RBE). With this approach, when

a virtual device is added to a VM running on host M1 and if the corresponding physical device is remote (present on M2), the LBE on M1 establishes a communication channel with the RBE on M2. The LBE then tunnels data between the FE and the RBE. In case of iConnect, the LBE also performs logical functionality as required.

The RVBD implementation follows the Netbus approach described above. In particular, the RVBD FE is similar to that of a normal VBD FE. When the RVBD FE inside the VM accesses the device by making requests over iConnect to its corresponding RVBD LBE running in VP at M1, the LBE converts these requests to remote access requests and forwards them to the RVBD RBE, which runs inside VP at M2. The RVBD RBE then makes the actual requests to the device and returns the responses to the RVBD LBE over the network. The LBE performs the logical translation of these responses to VBD responses, and in turn returns the VBD information to the RVBD FE over iConnect.

Computational results based on IOzone [18] benchmark presented in our previous work [86] demonstrate that a RVBD-based solution provides performance comparable to that of the NBD, with the added benefits described above. These experimental results use the driver domain-based realization of iConnect. We do not have a comparable SV-NIC based realization. This is because a TCP offload solution is not available to us for the IXP-based platform, but such a solution is required to implement the remote access component of the SV-NIC-based realization. To address this issue, we developed an alternative realization that uses message passing over ethernet to implement remote access. This 'lean' approach is in keeping with similar implementations done in the past [129] and with ongoing work in the high performance domain.

Table 2 shows initial results from this low latency implementation. It depicts the latency for implementing the logical block device functionality inside the SV-NIC based VP vs. inside the VM. In particular, we measure the time taken by SV-NIC to provide a RVBD response to the guest VM's block device driver, which includes the time taken by the driver to copy the data from the bounce buffers of network I/O to buffer cache pages. This copy is required due to the limited host memory accessibility of our current IXP-based board [111]. In contrast, for the VM-based implementation of this logical functionality, provided by NBD,

Table 2: Latency microbenchmark for providing a response to guest VM via RVBD and NBD. In both cases, SV-NIC provides either a RVBD or a VNIC, respectively.

	Latency (ms)/response	Interpolated latency (ms)
RVBD	0.323	0.323
NBD	2.01	0.89

the network packet is copied to the socket buffer from the bounce buffers by the network driver, and the rest is handled by the guest VM's networking stack. The application used for this benchmark is hdparm without prior caching of data, so as to measure the sequential read performance of the virtual disk. The average response size for NBD is 123570 bytes, while for RVBD, the average response size is 37601 bytes. We interpolate NBD's latency to a similar response size as that of RVBD. Results show that for response size of 37601 bytes, implementing logical block device functionality via RVBD provides a 64% latency reduction for a guest VM. The difference in latency is attributed to the removal of an extra copy (no need for movement via socket buffer to buffer cache) and the removal of networking stack. The SV-NIC implements the functionality by inspecting the packet header to identify the RVBD data, and handing it over to the logical block device driver running in guest VM. The cost incurred by the SV-NIC to implement this functionality is negligible ($\sim .3\mu s$) compared to the latency incurred at the host side.

4.3 Beyond I/O Extensions

The focus of this chapter is on iConnect's support for enhanced I/O for VMs, for which the abstraction presents opportunities to enrich VM-VMM interactions with new functionality. This section motivates the utility of this idea with additional examples of interesting iConnect functions:

- Operations inferring information about a guest VM's behavior, the purpose being to provide a VMM-level service that enhances a VM's execution experience on the virtual platform, while at the same time, optimizing the system's resource utilization. An example is inferring a guest VM's utilization of its buffer cache by looking at its page faults, page table updates, and virtual block device usage, using an iConnect

implementation that carries all of this information. This information has been shown useful for driving a VMM's memory allocation policies [79].

- 'Trust' operations that infer whether a guest VM's behavior constitutes malicious activities, where a departure from 'expected' behavior may warrant a reduction of trust by the overall system. An example of such iConnect functionality is described is Chapter 6.

- Implicit requirements by a guest VM that are necessary for its existence on the virtual platform. An example is the I/O emulation performed by the VP on a VT-enabled system in order to support fully virtualized guests. In this case, the guest VM sends data to a 'fake' physical device, which is intercepted and converted by the VP to a normal virtual device [110]. Another example is the hypercall interface and the exchange of semantically meaningful information exchange between VM and VMM, e.g., requests for page table updates [88].

4.4 Conclusions and Future Work

This chapter describes the iConnect abstraction that provides efficient and semantically enhanced communications for VMs. Focusing on I/O, it further describes different realizations for a VP, based solely on a VMM and on driver domains vs. a VP based on a VMM, on a self-virtualized device, and/or on driver domains. The VP implements useful functionalities/properties for *logical devices*, a key element of the software framework used to realize iConnect. We describe two such examples of logical devices – to provide QoS guarantees to a VM's network flows and to implement disk block level functionality to provide transparent access to remote block devices to guest VMs. Xen-based implementations of services providing these logical devices demonstrate substantial performance improvements and additional functionality derived from the corresponding logical devices at a minimal cost to VMs.

Prototype iConnect realizations based on SV-NIC can be further enhanced with intelligent use of the additional processing capabilities offered by high end NICs, leveraging our IXP NIC's self-virtualization functions. Specifically, the protocol for the block level access

over ethernet between the RVBD client (FE) and the RVBD server (RBE) can be made more reliable by using RUDP [51, 70] in SV-NIC. Also, by using jumbo frames, better performance can be obtained for large sequential reads. Another artifact of our implementation is that currently the RVBD server does not utilize any caching on the server end. This results in disk access for every read request, and adversely affects the latency contribution from the server end. An alternative is to interface RVBD server with OS at file system level, rather than disk block level.

CHAPTER V

VMEDIA: ENHANCED MULTIMEDIA SERVICES IN VIRTUALIZED SYSTEMS

This chapter presents the *VMedia* framework that provides the multimedia virtualization service for sharing media devices among multiple virtual machines (VMs). The framework implements a host-based iConnect realization and provides *logical* media devices to virtual machines. These devices are exported via a well defined, higher level, multimedia access interface to the applications and operating system running in a virtual machine. By using semantically meaningful information, rather than low-level raw data, within the VMedia framework, efficient virtualization solutions can be created for physical devices shared by multiple virtual machines. Experimental results demonstrate that the base cost of virtual device access via VMedia is small compared to native physical device access, and in addition, that these costs scale well with an increasing number of guest VMs. Here, VMedia's MediaGraph abstraction is a key contributor, since it also allows the framework to support dynamic restructuring, in order to adapt device accesses to changing requirements. Finally, VMedia permits platforms to offer new and enhanced logical device functionality at lower costs than those achievable with alternative solutions.

5.1 Introduction

With their ever-increasing processing capabilities, even desktop-class platforms can now sustainably execute the workloads imposed by multiple concurrent applications. For instance, a single high end home PC's resources can be shared to simultaneously play a video game, watch a movie, and perform financial tasks. In this chapter, we explore sharing opportunities and methods for multimedia devices, the goal being to make it easy to dynamically compose, share, and use these devices to provide efficient multimedia services. The concrete artifact resulting from this work is the VMedia addition to the Xen virtualization platform [48]. VMedia enables media-rich applications by better supporting flexible access

to and use of the many media devices present in today's systems. Specifically, VMedia offers new hypervisor-level support both (1) for efficient and flexible device sharing and (2) for dealing with and exploiting device differences and diversity.

Device virtualization is a key element of virtualized systems. A simple, non-intrusive method is to create a virtual device that emulates a physical one. In this case, the virtualized platform (VP) provides I/O resources (configuration registers/memory) just like the physical platform, and the guest OS interacts with the virtual device in the same fashion as it did with the physical device, using its own device driver. However, this approach has inherent performance limitations, because device emulation requires fine-grain involvement from the HV and/or Service VM (i.e., at the level of memory/register access). As an alternative, all current system virtualization solutions provide simpler virtual I/O devices, which present different access interfaces to guest VMs, such as shared memory circular buffer rings, rather than I/O memory and registers. Device drivers hide these interfaces from the guest OS kernel, providing it with standard device interfaces. For example, a virtual NIC device driver provides an ethernet interface that is identical to the interface provided by the physical NIC's device driver. Using these simpler virtual I/O devices, the corresponding device driver provides substantial performance benefits compared to the emulation approach. Above this layer, guest operating systems, then, operate just like in non-virtualized environments, using their device drivers and other internal functionality to present applications with efficient higher level system abstractions like sockets, files, etc.

For modern virtualized platforms, then, researchers and developers have already recognized that such virtualization requires guest OSes to use new device interfaces and drivers. Several interesting research questions result from this fact, including (1) whether there are enhancements of such interfaces useful to certain classes of applications, and (2) whether such enhanced I/O devices can be implemented efficiently or even used to realize performance improvements?

This chapter addresses these questions for multimedia systems and applications, by developing and experimenting with the *VMedia approach* to I/O virtualization. This approach

exports to applications *logical devices* that are semantically enhanced versions of the physical devices present in the underlying platforms. Specifically, a VMedia logical device has attributes and provides access methods that go beyond defining "what the device is", as in current systems, to also define "how it is used". For example, a logical camera device might provide a rich multimedia access interface, like Video4Linux [40] (V4L), instead of the low-level API presented by a USB camera. Other examples of these logical devices are described in Chapter 4.

Previous research has already demonstrated the utility of using logical rather than physical device interfaces. In our own work with V4L, for instance, we have shown that this interface can be used for transparent access to both local and remote physical camera devices [82]. The VMedia approach exploits I/O virtualization to go beyond such transparent device remoting: it provides a service-based interface to media devices in order (1) to allow sharing of these devices, and (2) make it possible to dynamically create from physical devices virtual ones with different properties and capabilities. We note here a similarity in approach between VMedia's service-based logical devices and modern file system services, such as NFS [28] and GPFS [121], provided by today's network storage solutions. Such file services can be seen as a logical device, which are provided in addition to block-based virtual disk devices (e.g. devices supporting SCSI interface). Utilizing filesystem level abstraction, these storage logical devices allow sharing of content (files) in a straightforward manner, while the usual block-based virtual devices do not.

Previous work has also shown the utility of using semantic information to enhance certain physical devices, as with smart disks [123], for instance. However, for cost reasons, these solutions have not been widely popular. VMedia addresses this issue by using software to enhance the virtual platform, rather than requiring new or extended device hardware (e.g., expensive device controllers). Furthermore, the service-based virtualization used by VMedia affords several additional advantages.

First, it can simplify the guest VM's OS kernel without sacrificing any of the functionality presented to applications. Second, by using domain-specific semantic knowledge, I/O virtualization at higher levels like V4L can provide better performance than solutions

operating at the device level. Third, the use of logical devices can provide better opportunities for consolidation in the Service VM, based on information from multiple guest VMs. Fourth, a logical device may provide better performance and/or more functionality than that offered by a single physical device, by having the Service VM use an ensemble of physical devices to realize the logical device, for example. Shifting logical functionality to the Service VM also frees up computational resources at the guest VM side. A guest VM can then use these resources to implement other useful functionality. It may also increase platform's scalability in terms of the number of VMs it can support. Shifting computations related to I/O also allows guests to function better in the presence of resource restrictions, such as limited availability of cores or licensing restrictions imposed by software for certain number of cores.

In summary, this chapter presents the VMedia framework for logical devices, focused on the multimedia domain:

- VMedia is used to export a 'multimedia' device to guest VMs, using the standard Video4Linux [40] interface.

- The multimedia device is implemented with software running in the Service VM, Dom0. By acting as a 'hub' for such logical virtual devices, the Service VM can provide enhanced multimedia services to guest VMs, with efficient and flexible device sharing, and offering new device capabilities. The Dom0-based realization of the multimedia virtualization service presented in this chapter is an example of host-based iConnect realization presented in Chapter 4.

Experimental results demonstrate the viability, utility, and performance of the VMedia approach and implementation. Multimedia device access can be performed by guest VMs via VMedia framework with low overhead ($\sim 0.25ms$ for an image capture of size 320X240). Using semantic information, VMedia's low overhead virtualization solution allows multiple guest VMs to share a media device with minimal overhead ($\sim 8ms$ for 8 VMs, each requesting images of size 320X240), as compared to the alternative method of semantic-unaware sharing via time-division sharing, which can impose overheads of upto 8X of physical device

access cost for 8 VMs. We also demonstrate the ability to implement logical devices as aggregations of multiple physical devices, again with very low overheads (0.21%).

5.2 VMedia Design and Architecture

VMedia Design. Unlike network and storage devices, which are virtualized via time- and space-sharing respectively, the rich semantics associated with multimedia devices make sharing at the device level more difficult. Web cameras and microphones, for instance, can be time-multiplexed among multiple VMs, but arbitration of the device will be difficult. For example, different VMs may want to change the attributes of the device in mutually exclusive ways. This means that the virtualization system must maintain a 'context' for the device per VM, and change the device to a particular context whenever the corresponding VM requests access. As a result, current virtualization solutions ensure that multimedia devices are used exclusively by one VM. Virtualization of these devices is done at a lower level, such as USB and PCI, and access is provided to a single VM as a passthrough.

The VMedia framework creates enhanced opportunities for sharing, by implementing logical devices that are accessed via a standard multimedia API, which is Video4Linux (V4L). Guest VMs' device drivers interact with the VMedia Service VM using a higher level API, again similar to V4L. VMedia's *virtual multimedia device* thus exported have several interesting properties. First, such a device need not be a simple mapping of the physical device that is being virtualized. In fact, additional interesting properties of a virtual device can be entirely implemented in software, an example being a virtual device that supports multiple palettes and image resolutions, while the physical camera supports only one of these. Second, device implementations can be entirely dynamic, using runtime code generation and extension techniques [68] and placing such extensions into Service VMs for shared use by all/some logical device users. Extensions may implement data transformations, for instance, to guarantee certain privacy constraints on the data captured by the device [82] or to provide data to end user applications in certain forms. Third and as explained next, multiple guests can efficiently share VMedia's multimedia devices, via its *MediaGraph* abstraction, described in detail in Section 5.2.3. The outcome is that a guest

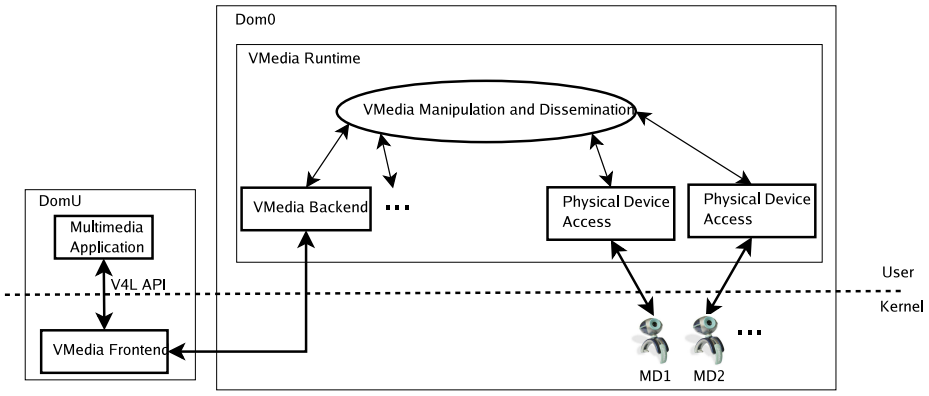


Figure 17: VMedia architecture.

VM can be oblivious to how the physical device is being accessed, and that end users need not rely on complex applications hosted by guest VMs for such purposes.

The VMedia design also makes it possible to compose new virtual devices from multiple, possibly heterogeneous physical devices. For example, by using two similar camera devices and with appropriate phase lag, it is possible to support twice the frame rate than what could otherwise be afforded by a single device. As another example, a context sensitive camera device can be created using a camera and a microphone where an image is only captured in the presence of sound, else returning an image from a cache without capturing a new physical image.

VMedia Architecture The VMedia framework consists of two main components: (1) virtual multimedia devices and associated drivers running in guest VMs (client side), and (2) the VMedia runtime that executes in the Service VM, or Dom0 (server side). The VMedia runtime accesses the physical multimedia devices and provides guest VMs with access to the media data via virtual devices. Figure 17 depicts a high-level overview of these components.

5.2.1 Client Side Components

Client (Guest VM) side components include a virtual multimedia device and the corresponding kernel device driver. The virtual multimedia device is an extension of the virtual interface (VIF) abstraction presented in Chapter 2, with similar API. The device is assigned

Table 3: Mapping between VM API and VMedia messages.

VM API	VMedia Message
open	SETSIZE, SETPALETTE
read	GETFRAME, (RECVFRAME)
VIDIOCMCAPTURE	SETSIZE, SETPALETTE, GETFRAME
VIDIOSYNC	(RECVFRAME)
VIDIOCSWIN	SETSIZE
VIDIOCSPICT	SETPALETTE

a *unique ID* and consists of two message queues, each of which is a circular ring buffer. One message queue, called the *send queue* is for outgoing messages to the physical device, sent from the guest VM to the VMedia runtime. The other queue, called the *receive queue* is for incoming messages from the device, sent from the VMedia runtime to the guest VM.

A pair of signals is associated with each queue. For the send queue, one signal is intended for use by the guest VM, to notify the VMedia runtime that the guest has enqueued a message in the send queue. The other signal is used by the VMedia runtime to notify the guest domain that it has received the message. The receive queue has signals similar to those of the send queue, except that the roles of guest VM and VMedia runtime are interchanged.

The kernel driver, called the VMedia frontend driver, registers a V4L device with the guest VM kernel. Applications running in the guest VM access the V4L device via V4L specific IOCTLs or file access system calls (e.g. read). These calls are converted into VMedia messages by the frontend driver, and sent to the other end via the send queue, where the backend component of the VMedia runtime receives them and performs appropriate actions. In response to these messages, the VMedia runtime may generate messages for guest VMs, which are received by the frontend via the receive queue. These messages in turn are mapped to application-specific calls. Table 3 shows the correspondence between key guest VM access API and VMedia messages. V4L-specific IOCTLs for VM API are capitalized. Also, VMedia messages in parentheses are receive queue messages, while others are send queue messages.

These messages do not carry media data themselves. All media I/O takes place via a pool of shared memory buffers shared between the guest VM and the VMedia runtime.

These buffers can also be mapped directly in application address space, thereby allowing I/O with minimal copying.

The virtual devices we have implemented to date are those focused on the multimedia domain, supporting properties related to a video capture device, such as image size, image depth and palette, via the V4L interface. Properties for devices other than video, e.g. audio and VBI, can also be provided via this interface, and this is part of our future work. Virtual devices also support some VMedia-specific logical properties, such as orientation and quality, exported to applications via an extension of V4L API. These properties, along with the multimedia properties discussed above, are used by the VMedia framework to compose efficient and enhanced virtualized I/O solutions. Improved performance coupled with transparency to applications and to the guest VM's operating system are the potential outcomes of this approach, as shown in more detail in Section 5.4 below.

5.2.2 VMedia Runtime

The VMedia runtime realizes the self-virtualized I/O abstraction [111] with software resident in a Service VM. The runtime is responsible for:

- scalable and isolated multiplexing/demultiplexing of a large number of *virtual devices* mapped to one or more physical devices;
- providing a lightweight API to the hypervisor and guest VMs for managing virtual devices;
- efficiently interacting with guest VMs via simple APIs for accessing the virtual devices; and
- implementing multimedia domain-specific extensions that enable semantically enhanced logical virtual devices.

These functionalities can be broadly categorized as 'management' and as 'I/O virtualization'. For a virtual multimedia device, management functionality is provided to the hypervisor and to the guest VM using the device. In addition to obvious management actions like device creation and removal, the VMedia runtime provides additional, domain-specific

reconfiguration functionality. For example, a video capture device may allow changes in image properties, such as colormap (color or grayscale), image depth and image size itself. The application running on the client side may request these changes, which in turn are sent to the VMedia runtime as management actions by the client side driver. The runtime makes appropriate changes in the properties associated with the virtual devices, along with any changes that may be necessary related to the I/O processing in order to satisfy these. For example, if the image size requested of a virtual multimedia device is different than that of physical device, an appropriate scaling filter may be installed in order to meet this mismatch. These reconfiguration actions are discussed in detail in Section 5.2.3.

The key functionality of the VMedia runtime is to implement I/O virtualization via sharing of physical multimedia devices among multiple virtual devices. The runtime utilizes semantic knowledge of virtual devices in order to perform this sharing. Since the runtime knows about the multimedia properties of the virtual device, e.g., the direction of I/O (input vs. output), type of content (such as image and audio), information about content (such as image size and colormap), it can use these properties in order to build an *information* flow from physical devices to virtual devices. For example, for input multimedia device, such as cameras, *images*, rather than *bytes*, are sent to virtual devices.

The VMedia runtime is composed of multiple entities that jointly realize the functionalities described above. These entities can be categorized broadly as (i) Physical Device Access, (ii) Virtual Device Backend, and (iii) Media Manipulation and Dissemination.

Physical device access entities implement the media device-specific methods for obtaining media data from or sending media data to the physical device, one entity per device. For example, the data could be obtained from the USB based camera via a V4L-based device driver, or it could be obtained over the network if the camera is attached to a remote device, such as a cellphone connected to the host system via USB, bluetooth, or wireless. Depending on the type of device and how it is connected to the host system, the latency and throughput of media data will vary.

Corresponding to every guest VM frontend, the VMedia runtime contains a *backend* entity. These form a point-to-point connection with the frontend, and merely work as a

gateway of information from (to) guest VMs to (from) VMedia runtime.

For input devices, such as cameras, captured media data from device access entities is provided to the VMedia manipulation and dissemination component (VMediaMD), where this data is transformed if required and is disseminated to the virtual device backend(s), which then flows to the guest VM frontend. For output devices, such as speakers, media data received from the guest VM is provided to the VMediaMD, where it is transformed if required and is provided to the appropriate physical device access entity for output. Currently, the VMedia framework only supports input devices, and hence, the remainder of this discussion is limited to these devices only.

The control flow for input multimedia devices (e.g., image capture requests and property changes) is similar to that of media data flow for output devices, with some exceptions. Management control requests may change the VMediaMD component itself. For example, if a virtual camera device requests a grayscale palette, the VMediaMD component may need to add another component to provide this functionality. Further, depending on the sharing of physical devices, if there is a common property/functionality required by all virtual devices and if it can be directly provided by the hardware, this control flow may reach the physical device access components themselves. Some of the management control decisions may only be taken at the service VM level itself, such as the orientation of the physical device.

I/O control requests go through a minimal path of the VMediaMD component, mostly providing arbitration. Arbitration decides which of the physical device access entities should receive this request (there may be more than one). VMedia-specific logical properties can also be used for arbitration. For example, a guest VM may indicate its preference for a certain viewing area via *orientation*. The arbitration logic matches this preference to one or more physical devices. Arbitration also decides whether it is necessary to forward a request to a physical device, since it may already be involved in the I/O. The request is only forwarded if it is not.

Media sharing in VMedia is governed by a simple arbitration principle – any request received from the guest VM during the time when a media capture I/O is pending can

be satisfied from the result of this capture. Hence, if multiple VMs issue capture requests simultaneously, the capture is performed only once and the result is distributed to all VMs. This type of device sharing is a special case of space sharing, where a device can be shared by all virtual devices at all times due to the *semantic* properties of the device.

5.2.3 The MediaGraph Abstraction

Abstractly, the VMedia runtime entities described above and the control and data flows implemented by them form a di-graph structure, termed MediaGraph. This graph is built to meet the properties specified by end user applications for the virtual multimedia devices they are using. Specifically, the MediaGraph implements efficient media dissemination by consolidating common computations and by reducing communication costs via data filtering. Moreover, the MediaGraph abstraction supports dynamic adaptation – it can be modified when new virtual devices are added and/or when the properties of existing multimedia devices are modified. Such modifications are triggered by configuration events generated by guest VMs and/or by monitored changes to devices.

Physical device access entities and virtual device backends form the edge vertices of the MediaGraph (sources and sinks, respectively), whereas VMediaMD entities form the internal vertices. These internal vertices correspond to various arbitration and transformation functions. Transformation functions perform the necessary conversions from the media format provided by the physical sources to formats desired by the backend at the guest VMs, and directed edges in the MediaGraph represent the control and data flows.

A sample MediaGraph is shown in Figure 18. As seen in the figure, the cameras, represented by the source nodes S_1 and S_2, generate image frames, that are then sent to the transformation nodes T_1 through T_4, that perform transform operations on the images and send the final outputs to the backend nodes K_1 through K_4, which provide the processed images to the multimedia guest VMs.

The MediaGraph abstraction enables efficient sharing of the multimedia content by avoiding redundant transformations that may be required by multiple sink nodes (backends) in order to support the properties. For instance, the graph shown in Figure 18 combines

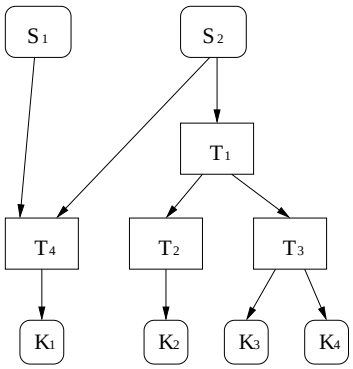


Figure 18: An example MediaGraph.

the common transformation T_1 for backend K_2, K_3 and K_4, thereby reducing the overall cost paid by the VMedia runtime. Next we describe an algorithm to maintain the efficient sharing when a sink node is added or deleted for a MediaGraph containing single source node.

5.2.3.1 Maintaining the MediaGraph for Efficient Sharing of a Single Source

We use a greedy algorithm to build and maintain the MediaGraph. Creation of a guest VM, and of the corresponding virtual media device, is translated to an addition of a sink to the MediaGraph. The desired content parameters of the sink are represented with an n-tuple $R =< r_1, r_2, ..., r_n >$. An example 3-tuple is <320x240, 8bpp, grayscale>, which corresponds to image resolution, image depth and color palette properties respectively. Starting with the source node (with content parameters $S =< s_1, s_2, ..., s_n >$), a check is performed to see if $R > S$, ($R > S$ iff $\exists i, 1 \leq i \leq n$, and $r_i > s_i$). Ordering for a specific property depends on the property itself (for instance, $8bpp < 16bpp < 24bpp$, among image depth). If this check succeeds, the source's parameters are updated to the maximum of the sink's desired parameters and the source's existing parameters. Next, all other transformations connected to the source are updated to reflect this change. The graph is then traversed in a breadth first manner to find a maximal match of the desired parameters among those of the existing transformations, and finally the sink is connected to the node with the maximal match, via any necessary transformations.

Deletion of a sink begins with removing the sink node from its parent and then traversing

towards the source node until all unnecessary transformation nodes (those that serve no other nodes) are removed. Finally, it is determined if the source's parameters can be lowered due to the removal of the sink, and if yes, carried out.

Changing a sink's parameters results in the actions of the deletion of the sink node from the MediaGraph, followed by an addition of a sink node with the new parameters. The MediaGraph maintenance algorithm and its evaluation are described in more detail elsewhere [112].

5.3 Implementation Details

The VMedia runtime is implemented as a user space application in the Service VM (Dom0) which completely encapsulates the I/O virtualization for the multimedia devices. Backend entities communicate with the frontends in guest VMs via Xen HV-specific communication mechanisms, which provide for the shared message queues and signaling. Different physical device access entities are run as separate threads to provide maximize concurrency in the runtime. These threads use device-specific methods for I/O. For example, for a USB based camera, the corresponding thread uses V4L IOCTL calls for image capture, similar to applications such as camE [5]. For a cellphone based camera, the corresponding thread communicates with a server process running on the cellphone that provides images over network.

For information dissemination between these edge nodes of the graph and to implement VMediaMD entities, the runtime utilizes an event-driven middleware, called EVPath [7]. EVPath allows data flow as *events* among nodes of an overlay termed *stones*. Stones can perform event processing, and can can transform an input event to an output event, possibly of different type, before passing it on to another stone. Stones can also perform routing decisions based on the event contents. This allows EVPath to perform content adaptation, which is required to support the logical functionality provide by VMedia.

The VMedia framework allows two types of logical functionality – one encoded in the V4L attributes of the virtual device itself, e.g. image size and colormap, the other completely based on guest VM. In the former case, the VMedia runtime installs well-defined

processing entities as stones in the MediaGraph. For example, if the image size of a virtual device is smaller than the physical device, a stone containing a scaledown filter is installed in the MediaGraph. Other filters, such as crop and grayscale, are installed in a similar fashion. VMedia also allows further predefined logical functionality via the extension of V4L attributes. For example, a virtual camera device may provide image data in specific image formats, such as JPEG and PNG. These functionalities can be provided in a manner similar to the earlier ones. These image processing-specific functionalities are implemented using the imlib [13] library.

5.4 Experimental Evaluation

We evaluate the VMedia framework on a desktop system with 3.2GHz dual-core Pentium-D processor and 3GB of RAM. To this machine are attached a Kensington SE401 USB-based camera, and a second Motorola e680 cellphone with a built-in camera. The e680 cellphone runs the Linux 2.4.20 kernel and is connected to the desktop via USB. The camera communicates with the desktop using the TCP/IP protocol, supported by a virtual network driver over USB stack. Service VM (Dom0) running VMedia runtime is allocated 512 MB RAM and one of the physical CPUs, and runs the Linux 2.6.16 kernel. Other CPU is shared among guest VMs, as determined by Xen's scheduling policy. The VMM virtualizing the desktop system is Xen version 3.0.3.

5.4.1 Overheads of VMedia Framework

This set of experiments quantifies the overhead of multimedia virtualization via the VMedia framework, measured as the difference between the latency of image capture experienced by a guest VM from the virtual multimedia device and the latency of image capture experienced by the VMedia framework from the physical device. This overhead includes the cost of transformations performed on the media data (computation), and its dissemination to virtual device frontends (communication). The content is delivered only to those virtual devices that request it, even if these virtual devices share some (or all) of the VMediaMD components with other devices that did not request it. For these experiments, the image properties (size, palette etc.) for virtual and physical media devices are kept the same, so

the only overhead incurred is due to dissemination.

The scalability of VMedia is demonstrated by increasing the number of VMs and measuring the *amortized overhead*. As number of VMs are increased, transmission costs of VMedia runtime increase as media data needs to be disseminated to more and more VMs. However, the latency of image capture as experienced by a guest VM depends on the amount of sharing, as the cost of a physical I/O gets amortized over multiple virtual I/O requests. To capture this sharing effect, we only account for the net positive overhead experienced by a guest VM, which includes VMedia's dissemination cost. We average over all net positive overheads experienced by N VMs sharing a physical device, and report it as amortized overhead. The overall cost of virtualization also increases due to scheduling, since context switching of VMs is required on a single CPU. In future multi- and many-core systems, the scheduling costs will be smaller, or even negligible, if there are enough physical CPUs.

We compare the VMedia overhead with a *time-sharing* approach of virtualizing the multimedia device. In this approach, every guest VM image capture request results in a image capture from physical device. In the presence of no contention, this approach is comparable to VMedia. However, in case when multiple VMs require access to the media device, the overhead of this approach not only includes communication of media data from the service VM, it may also include image capture latency from physical device for another VM. The overhead of this approach, hence, is always positive, and we report the average overhead per request.

We evaluate both VMedia and time-sharing approaches in two scenarios. In one scenario, termed 'no-wait', VMs successively request image capture from virtual devices without any wait between them. In another scenario, termed 'random-wait', a VM waits a random amount of time between $[0, 1000]$ milliseconds before making another request.

Figure 19(a) compares the overheads of VMedia and time-sharing approaches. For a single VM, the overhead of both the approaches are negligible. However, as the number of VMs increase, the overhead of time-sharing approach increases rapidly, including multiples of physical capture time as a factor, in both 'no-wait' and 'random-wait', with latter being slightly better than the former. The overhead of VMedia approach also increases, but only

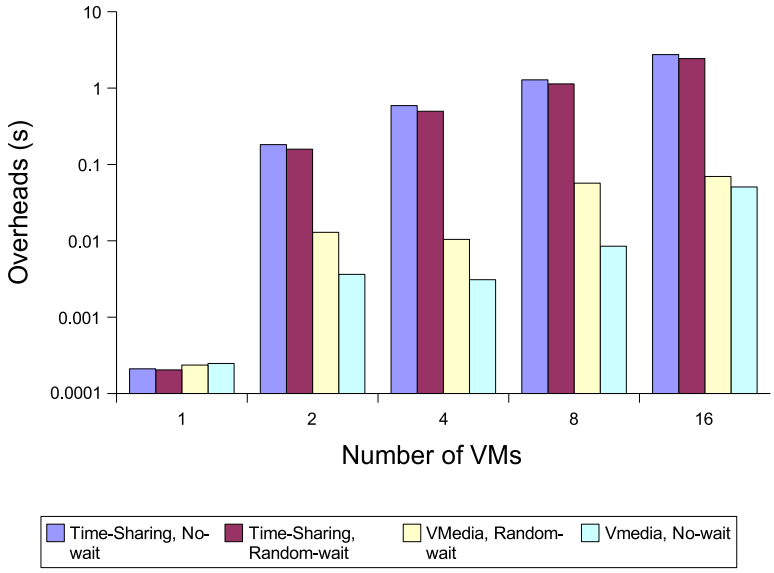
due to the communication cost of VMedia and context-switching cost of VMs. Both of these overheads are small when compared to the physical capture time. The overall overhead of I/O for an image capture from the virtual device with increasing number of guest VMs becomes as high as $\sim 25\%$ of the overall virtual device capture cost.

For each scenario, we also present the *sharing factor*, which demonstrates the underlying approach's ability to share the device, the higher the better. This factor is calculated as $\frac{\sum_{i=1}^{N} \text{captures from virtual device i}}{\text{captures from physical device}}$, N being the number of guest VMs. Figure 19(b) compares the sharing factor for different virtualization approaches in two scenarios, as mentioned earlier. For perfect sharing, the sharing factor should increase linearly with increasing number of guest VMs. However, due to high context switching costs, we observe the best case sharing factor to be ~ 8. The sharing factor of time-sharing approach is always 1, since every virtual capture request results in a physical capture request. The VMedia approach attains best sharing factor for the 'no-wait' case, while the sharing factor reduces as the contention for the physical device is reduced in the 'random-wait' case.

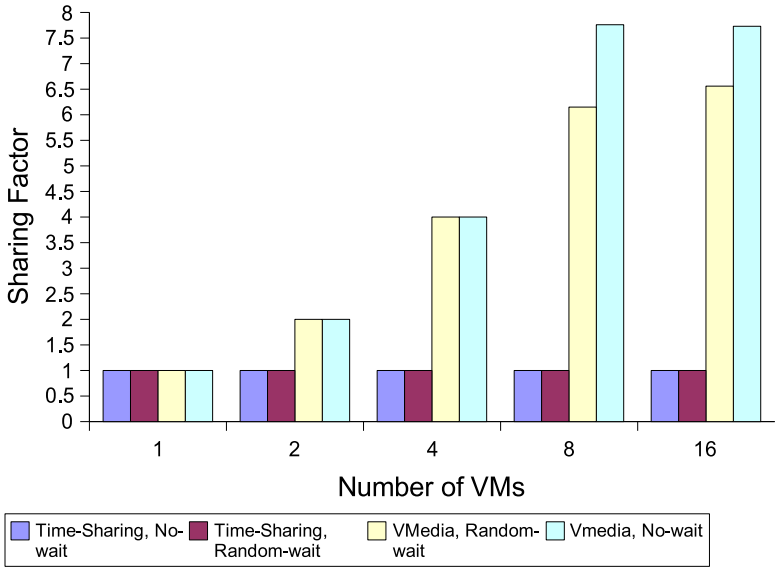
These results show that the VMedia framework shares physical devices efficiently, which in turn contributes to its performance and scalability. Further, using higher level 'V4L' requests, the no-sharing passthrough type virtualization for a single VM can be achieved at a lower cost than, e.g., using USB level requests [86] – where every single USB level request adds an overhead of about 25%.

5.4.2 Enhanced functionality sharing

Results in the previous section demonstrate the performance benefits derived from device sharing and the consequent amortization of I/O costs. However, using MediaGraph, the VMedia framework affords further benefits by sharing at the *logical* level. To demonstrate the benefits of enhanced sharing via the MediaGraph, we construct the following scenario. Four guest VMs are created in Xen – two VMs, VM1 and VM2, require images of size 640x480, while VM3 and VM4 require images of size 160x120. VM4 also requires grayscale images. We compare two approaches to sharing – the naive way, where the VMedia framework only shares the physical device and any transformations are performed by the guest



(a) Overhead, results are reported on log scale on y-axis.



(b) Sharing factor.

Figure 19: Comparative evaluation of VMedia and time-sharing approaches.

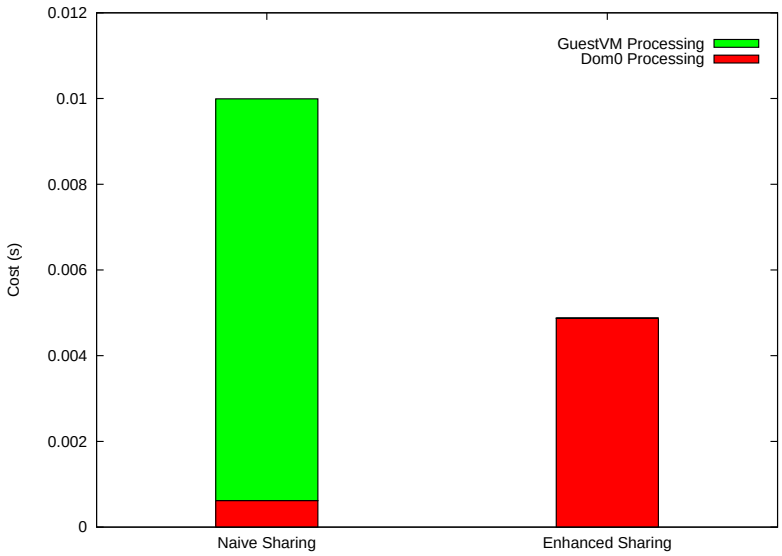


Figure 20: Comparison of enhanced sharing vs. naive sharing.

VMs themselves and the enhanced way, where the VMedia framework also performs any required transformations. These transformations are derived by the framework based on the *parameters* of the virtual devices, namely image size and color palette.

In this case, since the MediaGraph reduces the amount of redundant transformations performed, we expect to see lower processing costs. As shown in Figure 20, since all transformation-related processing is performed in the Service VM with the MediaGraph, we see a higher cost. In the naive sharing case, the images are simply sent to all VMs. However, the guest VMs perform all of the transformations in the latter, which is completely absent in the former. The overall costs, as shown in the figure, are almost 50% lower due to elimination of redundant computation.

5.4.3 Dynamic Restructuring of MediaGraph

In this section, we quantify the overheads of VMedia framework associated with dynamic restructuring of MediaGraph. Restructuring is performed in response to the management actions performed on the virtual media devices. These actions include opening and closing of devices, and changing their properties, such as image size and color palette, via IOCTL calls. The framework translates these actions into MediaGraph modifications, as described earlier in Section 5.2.3. The modifications require creation and removal of nodes from the

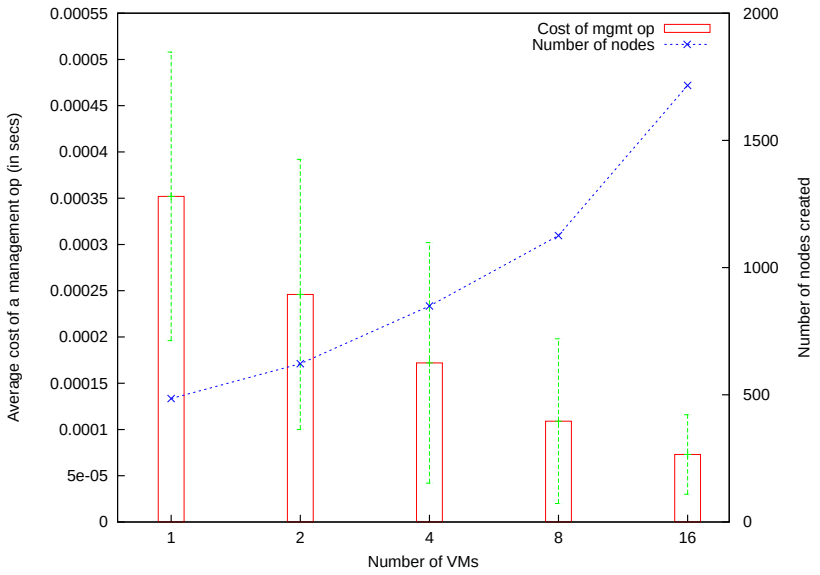


Figure 21: Management cost of VMedia runtime.

graph, which in turn require EVPath *stones* to be added/removed.

We measure the cost associated with a management action as (1) the time it takes VMedia runtime running in Dom0 to carry out the modifications, and (2) the amount of change in the MediaGraph resulting from these modifications. Since the removal of stones takes significantly less time compared to their additions, we only consider the number of stone additions as the metric for the amount of change in the MediaGraph. Figure 21 depicts these results. On x-axis, we vary the number of VMs (and hence the number of virtual devices). Each VM performs 100 management actions related to device property changes. Each action is drawn randomly-uniformly from a set of 5 such changes - 3 related to image size changes and 2 related to color palette changes. Each VM also waits for a random amount between $[0, 1000)$ milliseconds, between two consecutive actions.

Results demonstrate that with increasing number of VMs, the average cost per action decreases, since the cost of MediaGraph change could be amortized over actions from different VMs. Also, the amount of change required for MediaGraph increases sub-linearly. Put differently, the amount of change per management action decreases with increasing number of VMs. This explains the decreasing average cost of management actions.

Table 4: Cost components for multi-camera aggregation via concatenation.

	Cost (ms)	Cost (% of Vdev Cost)
Vdev Capture	622.023	100
Phys Capture	619.624	99.61
Transformation	.907	0.15
Communication	.366	0.06
Miscellaneous	1.127	0.18

5.4.4 Enhanced Logical Devices via Multi-Device Aggregation

Depending on the requirements of a guest VM and the availability of physical devices, certain services can be composed that allow a guest VM *improved* quality of service. For example, if a guest requests a wide image (of aspect other than regular 4:3), VMedia can aggregate images from multiple cameras either horizontally and/or vertically. Similarly, if a better resolution image is required, e.g. 640X480, but physical cameras can only provide 320X240, four such cameras can be aggregated. This is better than just using scaling – since it does not result in any quality loss. This can be further extended with additional processing to create a video wall [136]. Table 4 shows the microbenchmarks for a virtual camera device created by the concatenation of two cameras, the USB camera and the cellphone camera. The cost of physical device capture is taken as the maximum of these two devices, which corresponds to the cellphone camera. VMedia framework overhead includes the concatenation transformation action and the communication cost, and is very small compared to I/O latency.

Another example of aggregation is to use multiple media capture devices, possibly with a phase lag, in order to minimize the average latency of media access to guest VMs, where these devices are sampling the same environment. For example, for a single continuous image source with interframe latency L, average latency for capturing an image is $L/2$ – assuming accesses arrive randomly over a uniform distribution. However, by using two such image sources, and running then with phase lag $L/2$, the average latency can be reduced to $L/4$, effectively doubling the frame rate. To demonstrate the viability of this approach, we use two cameras, one USB and one cellphone camera, to capture frames in parallel, in a time period T, and timestamp them. The latency of frame capture from USB camera is $\sim 200ms$,

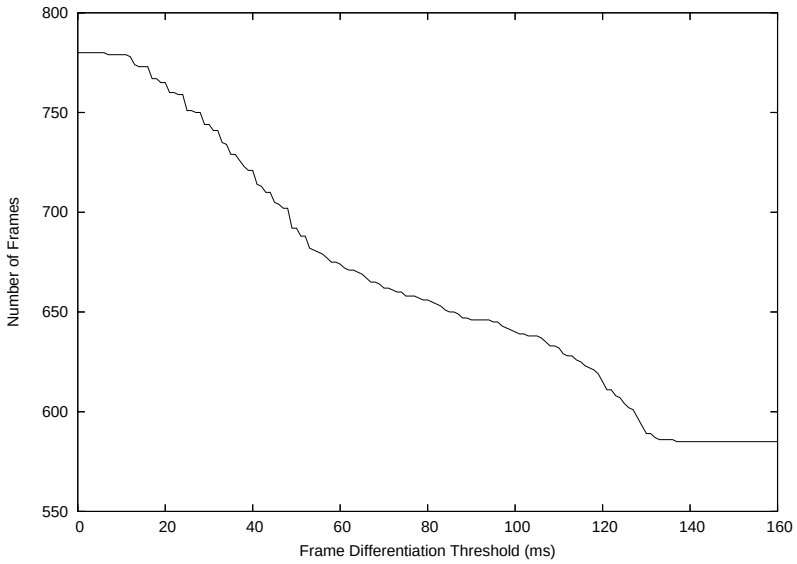


Figure 22: Number of distinct frames from two cameras in response to changing frame differentiation threshold.

while from cellphone camera, it is $\sim 600ms$ ($\sim 300ms$ of which is the core physical capture latency on cellphone and rest of it is the network transfer over USB to desktop machine.) Next, we coalesce i'th frame from device 1 ($f_{i,1}$) and j'th frame from device 2 ($f_{j,2}$), iff $|T_{f_{i,1}} - T_{f_{j,2}}| < \delta$, where δ is the frame differentiation threshold. This threshold quantifies the difference in media content, and hence the value, provided by successive frames captured by different devices. At lower values of frame differentiation threshold, the added value of extra frame is less. The resultant number of frames denote the *valuable* content. We plot the resultant number of frames obtained in a 2 minute time-period against different values of δ, as shown in Figure 22. The result shows that the aggregate device can achieve more distinct frames than a single camera, and hence can provide better frame rate to the clients. Note that for high values of δ, the number of distinct frames are small, and asymptotically reach the number of frames provided by the faster device (~ 580 in this case), thereby limiting the benefits from using multiple devices. For lower values of δ, the number of frames are larger, although the difference in media content may be smaller, again limiting the benefits from using multiple devices.

Alternatively, such services can be created in the guest VM itself – if we provide one virtual media device per physical device. This can be accomplished, e.g., by the VMedia

framework itself, by creating multiple MediaGraphs, one for each physical camera. The passthrough access to physical devices can be utilized in a similar fashion. As argued earlier, the latter approach does not provide sharing, and hence is of little interest. We believe that a single MediaGraph with support for aggregation is better than aggregation in guest VMs, for the following reasons:

- The guest VM implementation couples virtual devices with the physical environment, e.g., in number of devices and their orientation. This is usually a concern in virtualized environments, since a VM may be migrated to a different physical platform. Hence, the service implementation on guest VMs must be able to adapt to any changes in the physical platform. This adds complexity for VMs. By keeping this functionality purely in the VMedia framework, the framework – local to a single physical platform – provides a better way to provide this service.

- A single MediaGraph allows for enhanced sharing, in case aggregation is utilized by multiple VMs. Computations for logical functionality can be performed once, and results can be shared among multiple guest VMs.

5.5 Conclusions and Future Work

Efficient multimedia device virtualization and sharing requires that these devices be virtualized at a higher, 'semantic' level, rather than the traditional approaches, which incur high overheads. This can be obtained via the virtualization services approach, where logical devices share semantic knowledge with the Service VM that virtualizes the device. The VMedia framework implements such an approach, for virtualizing multimedia devices among multiple guest VMs. The framework also allows further benefits via enhanced functionality sharing, and it can potentially reduce the overall cost of multimedia services provided to guest VMs. Experimental results demonstrate that the overhead of the VMedia framework is small, and that it scales well with increasing numbers of virtual devices and virtual machines. The framework also supports dynamic adaptation in response to the changing demands of guest VMs, communicated in terms of virtual device properties and guest specific computations, and enhanced virtual devices with new and interesting functionalities

via aggregation of multiple physical devices.

The VMedia framework can be extended to include devices from multiple systems in a distributed environment. This requires that the MediaGraph composition and restructuring span multiple nodes. The framework can also be extended to include other types of devices, such as sensors and storage devices. Using multiple heterogeneous types of devices provides opportunities for interesting functionality that could be exported to guest VMs by the Service VM. One such example is *context-aware storage*, where context is derived from media devices, and based on that, access of certain content is performed from a specific storage device.

CHAPTER VI

O2S2: ENHANCED OBJECT-BASED VIRTUALIZED STORAGE

Object based storage devices (OSDs) elevate the level of abstraction presented to clients, thereby permitting them to offer methods for managing, sharing, and securing information that go beyond those offered by block-based stores. The Object-Oriented Storage System (O2S2) architecture presented and evaluated in this chapter implements a virtualization service to provide object-based storage in a virtualized environment. This service provides a virtual object-based storage device (vOSD) to virtual machines. The use of vOSDs permits the service provider, i.e., the *vOSD storage domain*, to offer to guest virtual machines new methods for resource management and consolidation, without requiring the purchase of physical storage devices that faithfully implement OSD functionality. Methods demonstrated in this chapter include improved support for access control and for heterogeneity of storage devices. Advantages derived from such methods also include reduced complexity for end clients, i.e., guest VMs. A prototype PVFS-based O2S2 implementation demonstrates that its enhanced services can be provided at low cost, enabled in part by the efficient utilization of otherwise idle domain resources.

6.1 Introduction

Storage virtualization is a mature area of computing, including commercial solutions, such as IBM's System Storage DS8000 and EMC's Centera. Such 'storage appliances' are in common use in well-networked environments like data centers, but low cost implementations have even enabled them for personal/home systems [8]. Their realizations utilize technologies like Network Attached Storage (*NAS*) and Storage Area Networks (*SANs*) to provide end clients with virtualized storage devices, where NAS and SAN technologies differ in the interfaces they provide to the end client. A SAN solution provides low-level block-based storage access, while a NAS solution provides higher-level file-based access. Abstracting from these differences and for simplicity, when referring to NAS or SAN, this chapter terms

the entity implementing any such virtualized storage solution a *storage domain*.

Any storage domain must answer multiple questions, including (1) what is the access interface provided to the client – block based vs. object (file) based, and (2) how is the data stored on (mapped to) physical devices? Depending on the answer to (1), the storage domain has different degrees of freedom concerning how to store the data. For example, if the interface is block-level, any storage decision must be made at that granularity, which also means that the client has the obligation to make such decisions, thereby increasing client complexity and reducing flexibility for the storage domain. Further, because of the large overheads of maintaining metadata about every single block, the storage domain is typically agnostic of the properties of the actual data stored, also implying that useful device properties like fault tolerance and striped I/O must be realized at the virtual block device granularity. In contrast, improved solutions are possible when allowing properties to be maintained on a per object basis, where an object-based interface provides opportunities for new reliability methods [141], for increased scalability [142, 62], for increased resource consolidation, and for data sharing among multiple clients [65].

Following the paradigm of object based storage, this chapter presents the design and implementation of an Object-Oriented Storage System (O2S2) architecture that can provide virtual object-store devices (vOSDs) and services to any virtual machine via the *vOSD storage domain*, without the need for specialized object storage hardware [24]. Moreover, these vOSDs can provide *semantically enhanced – logical –* object storage. That is, for any object stored by a client on a vOSD, the latter can store additional attributes (i.e., metadata) such as provenance [35], consistency [101], and client-specific information pertaining to the semantics of its content (e.g., an object containing a 'health' record or one that contains multimedia data). Accordingly, some of these vOSD object attributes will be solely managed by the vOSD storage domain, in a client-oblivious manner, while others are shared between the clients and the domain. Regardless of how such management is performed, however, it is these attributes, along with the attributes of the physical devices being managed by the vOSD storage domain, that permit the domain to provide enhanced functionality and services to storage clients. In contrast, more traditionally, a vOSD stores

raw data like that associated with files in a filesystem and limited semantic meta-data associated with these files, such as primitive access control information, size, type of data, and useful timestamps.

Beyond presenting the O2S2 architecture and its prototype implementation, our research also explores new and useful functionality associated with vOSDs. One class of such functionality, focused on the way objects are stored, concerns exploiting the different properties of physical storage media. The idea is that by aggregating an ensemble of physical devices, a storage domain can often provide better performance or enhanced functionality to the virtual device than by using a single physical device [122]. In addition, semantic information about the objects being stored can be used when aggregating physical devices. For example, an object-based storage system may provide striping and RAID functionalities only to the objects that actually require them. As another example, the system may provide *differentiated storage*, where the mapping between an object and a particular physical device can be decided based on object or device attributes, such as those pertaining to privacy and mobility constraints. In fact, such *semantic aggregation* is not unique to storage devices. Its use with camera devices, for instance, makes it possible to seamlessly join video streams from multiple cameras in order to provide a virtual video wall [136]. We note that semantic aggregation cannot be done with SAN solutions or with lower level device aggregation like that provided by the Logical Volume Manager [34]. This is because it is the object based interface and the object attributes that enable semantic aggregation at object granularity.

Another class of functionality enabled by the O2S2 architecture is fine-grained, object-based, access control. Based on the *labels* of objects and the clients who access them, the vOSD storage domain implements *Role-based Access Control* (RBAC) [66]. Enforcing access control at object granularity provides sharing and consolidation of resources superior to that offered by the large storage partitions present in block-based systems. Further, the storage domain can be integrated with the *trust management* component of a platform, where it can utilize 'trust' related information about a client to enforce *dynamic* RBAC. Additionally, such access control enables object-level logging of a client's accesses, which can be used to enhance security.

An important element of the O2S2 architecture is its use of a 'storage domain'. This provides independence from specialized storage hardware, such as object stores [24] and other enhanced disk-controllers [146]. In addition, the object properties implemented by a storage domain and desired by end users can transcend what is offered by backend hardware. Examples include encryption or secret sharing [125] techniques and transformation of data in client-specific ways [92], such as for obfuscation and specialization [149] purposes.

Finally, while vOSD storage domains can be constructed in many ways, this chapter describes domain realizations geared to meet the challenges of virtualized execution environments. In this context, vOSD storage domain clients are *Virtual Machines* (VMs), which execute on a Virtualized Platform provided by a hypervisor or Virtual Machine Monitor (VMM). Additional Service VMs implement virtualized services for these VMs. The vOSD storage domain implements the storage service inside a Service VM; other examples include 'Dom0' providing network virtualization [110] and the VMedia runtime [113] providing multimedia device virtualization. Experimental evaluations presented in this chapter, therefore, are carried out in the contexts of VMs, VM usage of the vOSD storage domain, and the VM-level overheads experienced in these settings.

Experimental results based on a PVFS-based prototype implementation of O2S2 architecture and its realization of vOSDs demonstrate (1) that the cost imposed by enhanced vOSD functionalities is low and (2) that such functionality scales well with an increasing number of client VMs. In particular, the cost of per-object access control is 2.5% and .4% for large reads and large writes, respectively, as compared to the case where no access control is enforced. Also, by using the resources available at the vOSD storage domain, it is possible to obtain performance benefits of $\sim 3X$ for large reads, as compared to current virtual block based storage solutions in the Xen virtualization environment.

6.2 Motivation

This section describes the scenarios that motivate the design of the O2S2 enhanced storage architecture. It describes how this architecture enables improved performance, better resource consolidation, easier trust management, and increased usability in heterogeneous

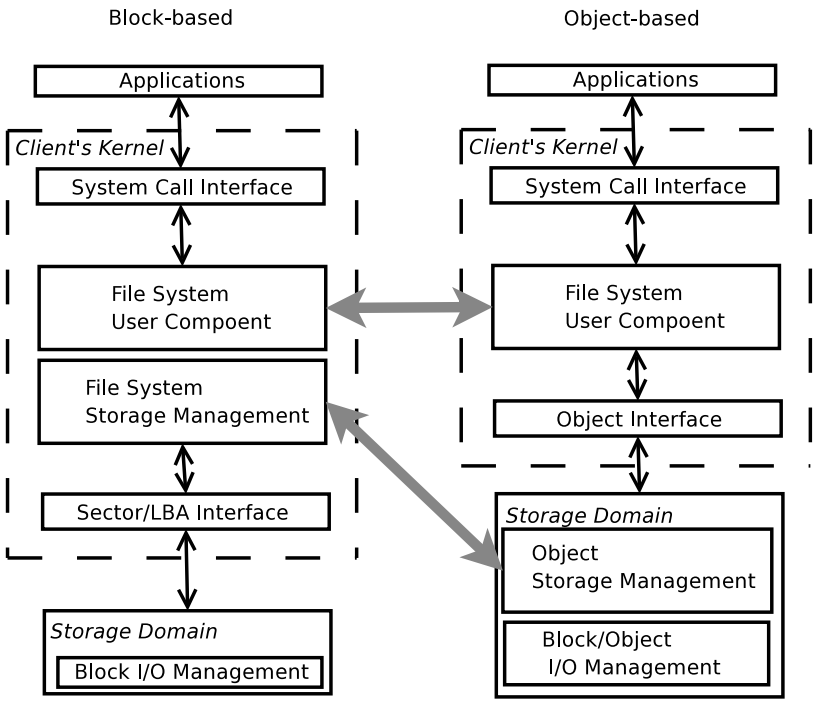


Figure 23: Comparison of block- and object-based interfaces for storage clients.

storage environments.

6.2.1 Object-based Storage Interfaces

Object-based storage interfaces, like those presented by the file-based interfaces of cluster or distributed file systems (e.g., PVFS [55], LWFS [101], Lustre [21] and Coda [52]), and by object storage devices (OSDs) [140, 72, 65], provide notable benefits for the storage client. For instance, when the tasks of storage allocation and access control are delegated to the storage domain, this simplifies the client's kernel in that it is only required to run a minimal file system. Further, since operating systems already maintain information, both data and meta-data, grouped at file-level, object-based storage interfaces provide an appropriate match between the capabilities of the virtual device and the requirements of the client. Figure 23, derived from [140], highlights the differences between an object- and a block-based interface. vOSDs and the O2S2 architecture underlying them exploit these differences to maintain and use novel meta-data with storage objects, as explained in Section 6.2.2.1.

The vOSD storage domain benefits from the presence of object based interfaces, because

storage management can be performed at a semantically meaningful level. This facilitates sharing and presenting opportunities for resource consolidation, and more importantly, it provides opportunities for *enhanced* virtual storage, based on additional per-object metadata, at costs that scale with the number of objects rather than with object size (i.e., number of blocks in an object). Further, since the type of storage devices used by vOSD storage domains is orthogonal to the interface provided to clients, the implementation of vOSD does not require physical OSDs. The use of such new physical devices can reduce the costs of storage management in a SAN environment, since host-resident space allocation functionality of a storage domain is no longer required. On the other hand and as shown in this chapter, a resource-rich realization of a vOSD storage domain can exploit the additional compute resources available at host-level to provide new and useful storage functionality to end clients and/or to shift certain computational tasks from clients to storage domains. Idle cycles typically available on storage domains [108] demonstrate the viability of this approach, as discussed further in Section 6.3.

6.2.2 Storage in Virtualized Systems

The realization of the O2S2 architecture presented in this chapter is based on common methods for platform and storage virtualization. Stated explicitly, it *satisfies the storage needs of virtual machines as storage clients* by utilizing a separate storage domain onto the same physical platform as the guest. This architecture affords the common advantages ascribed to virtualized resources, including improved resource consolidation, better isolation across different applications, and reduced vulnerabilities in personal/home environments. Additional advantages include improved manageability, as argued by Chandra et al [58]. In contrast to current virtualization systems [48, 130], however, our approach uses vOSDs to replace the block-based storage interfaces, such as IDE and SCSI, currently being used. The intent, of course, is to attain the goals of improved manageability and enhanced functionality articulated earlier.

6.2.2.1 From Virtualized to Security-enhanced and Trusted Object Stores

In order to maintain security isolation among multiple vOSDs, the vOSD storage domain employs object-level access control. This allows sharing physical storage space among multiple vOSDs at an object granularity. This access control is Role Based Access Control (RBAC) based on *labels* or *roles*, which define the *capability* of a storage client. The storage domain maintains these labels for all the objects it stores, and utilizes an external trusted entity for the management of labels associated with a storage client. This enables the storage domain to provide access control functionality at a reduced cost.

Another key advantage of our virtualization-based realization of vOSD storage domain is that it can monitor, inspect, and manage guest VMs and their use of storage 'from the outside', using privileged domains that are not subject to the same attacks or failures faced by guests running standard operating systems and applications across open network environments. These privileged management domains or 'trust controllers' can use VM introspection (e.g., using the XenAccess [41] facility developed in our research), behavior monitoring (e.g., as done in our work on power management [99]), or I/O traffic monitoring to continually assess VM 'health' or security [105]. Such domains can even intercept I/O requests to implement new security services like firewalls or intrusion detection [30] or to re-direct certain VM actions to guarantee desired safety properties for potentially unsafe code [75].

Leveraging the abilities of access control and external monitoring provided by system virtualization, the O2S2 architecture permits storage domains to enforce desired access controls on their data. This is done by using online monitoring to establish certain 'trust' values for guest VMs and for the platforms on which they run [134], and then, enforcing access controls to ensure that data requiring certain levels of trust is accessed only by those VMs on those machines that meet those requirements. Stated differently, these 'trust' values can dynamically change the *labels* or capabilities of a client. Even though a client's label might match the label of a particular object, a dynamic label based on the original label and 'trust' value might not, resulting in declined access.

Underlying these online matching processes, of course, are basic actions taken by storage

domains that (1) track (i.e., monitor) and label (i.e., compute and maintain trust-relevant metadata) the data items written and read by certain guest VMs, and (2) enforce that data items are stored and accessed only when trust values match, as per the access control or security policies stated by system administrators. A sample use case for such functionality considers doctors or nurses who create and access patient records. Here, records are labeled as per the sources that produce them, accesses require appropriate identities or roles, and in addition, they require that such accesses are only carried out from trusted platforms and guest VMs. The vOSD storage domain enforces policies like these by checking the labels (i.e., metadata) associated with data objects like patient records against the trust values of the guest VMs performing such accesses. It also enforces such properties for record storage, for instance, that certain patient records are stored on disks present at a location with better physical security.

Figure 24 shows the relationship between a distributed application running inside application VMs and a vOSD storage domain. The storage domain is a trusted entity and enforces access control itself, with the help of certain security extensions in the hypervisor. The application's behavior contributes to its "trust" value, as perceived by the trust controller and exported to the storage domain. If some part of the application runs on an untrusted platform, its "trust" value must be defined by a remote trust controller running on a trusted machine.

Fine-grained access control and metric like 'trust' are particularly relevant in data center settings, where their use extends the measures used for service level agreements (SLAs), which are typically defined to provide *statistical* guarantees on various performance characteristics of services, such as bandwidth and latency [57]. This extension is important because with multiple compute VMs [2] or storage services [3] hosted by the data center, service clients can no longer control those services' uses of data center resources. In this context, a secure and trusted vOSD storage domain can provide strong guarantees to a VM that houses sensitive information (e.g., patient records and proprietary art work) that this information will only be stored on some few *identifiable* disks, thereby reducing the risk of data being stolen or being retained after the client's run has completed. Further, when

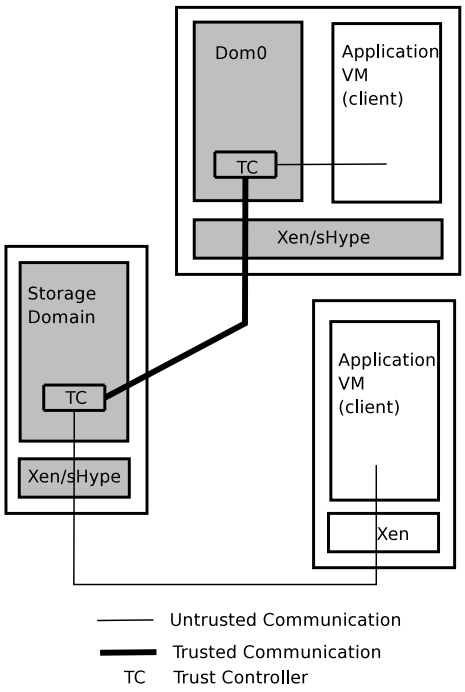


Figure 24: Storage domain as a trusted object store. Entities in gray are trusted.

upholding the integrity of a client's actions in a virtualized environment, if data is erased by the client, the service provider must ensure that none of this data is left anywhere in the system. This can be achieved with an improved accounting by storage domains about which client's data is stored on which physical media. Such accounting functionality can also help with data recovery upon loss, media recovery, and similar tasks.

Finally, a trusted vOSD object store can be used to enhance trust management itself. In particular, a VM's access log maintained by the vOSD storage domain can be used to generate a *behavior profile* for the VM at object granularity. Such profiles are easier to manage and more scalable than those based on block-level information [105], since monitoring need not deal with client-specific information, e.g., the file-system layout. Behavior profiles can then be used to derive 'trust' values for the VM. Also, in case of a security concern, such as a world-wide virus spread, these profiles can be quickly disseminated, as signatures, to preemptively stop damaged domains from being run or used, until the problem is corrected.

An extension of the security-enhanced vOSDs presented in this chapter concern the

auditability of such assurances. This may require specialized hardware such as Write-Once-Read-Multiple (WORM) devices [138] and an open logging infrastructure with access provided to clients. With such support, clients can then corroborate the actions taken by the service provider in response to their own actions, and they can ensure that the identities of the physical devices used match the ones enforced by the SLA. Furthermore, immutable content on WORM devices can be upheld legally in case of disputes.

6.2.3 Usability in Personal/Home Environments

As stated earlier, the implementation of a vOSD does not rely on specific storage media or subsystems. This affords us with substantial advantages in environments that employ diverse storage devices and where device usage depends on dynamic measures like current context. In home or personal environments, for instance, examples include a user storing media files on a video/mp3 player, personal contact information on a cell phone/PDA, running applications from local hard disk, and archiving information on a high capacity USB hard disk and/or on DVD media.

The O2S2 architecture can easily exploit diverse devices used in dynamic settings, where based on the contextual properties of each object designated by its owner and that of storage devices, the vOSD storage domain finds the best match for storing an object in a client oblivious manner, thereby relieving application VMs from making these decisions. One method is to store objects based on their performance metrics and/or on the performance properties of storage devices, in a manner similar to that of Stonehenge [76]. Another method uses access control based on labeling, which makes it possible to consider personal devices as potential storage media, by ensuring that certain data is stored and/or accessed only on certain devices. As a concrete scenario, consider user 'A' who has connected his iPod to her home PC. The iPod is labeled with the contextual label 'media' and with the access control label 'VM A'. The virtual machine owned by 'A' is also labeled as 'VM A'. The vOSD storage domain, then, makes the content of the iPod available via the virtual disk for 'VM A'. Also, if 'A' downloads a multimedia content from the internet and sets its contextual label to 'media', the content will automatically be stored on the iPod. From

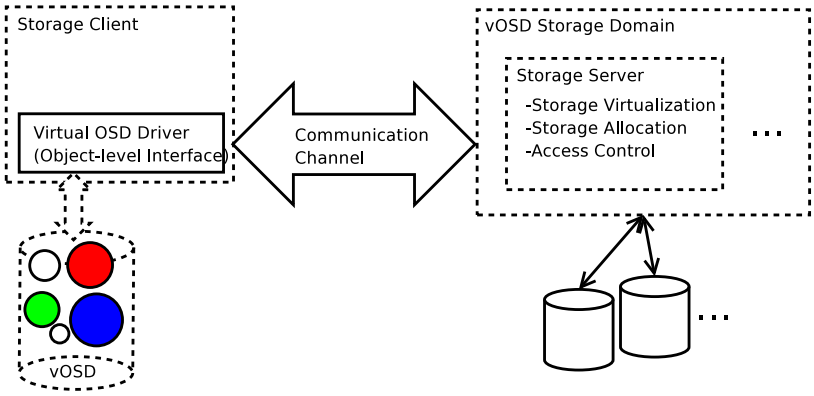


Figure 25: O2S2 architecture.

then on, this content will be available to 'A' 'on the go'. In contrast, all temporary files created by 'VM A' are stored on the generic hard disk. In this manner, the vOSD storage domain performs the semantic aggregation of hard disk and iPod to present a single vOSD to 'VM A'.

In summary, the O2S2 architecture provides enhanced storage for clients in virtualized environments. A key component of this architecture is its object-based interface for virtualized storage access for client VMs, providing benefits that include improved resource consolidation, better usability in heterogeneous environments, and object-level access control and trust management.

6.3 Architecture

The Object-Oriented Storage System (O2S2) has three main components:

- client-side virtual object-store device (vOSD) and the associated access interface provide to guest VM.
- communication channel between virtual object-store device and storage domain.
- the vOSD storage domain.

Figure 25 depicts relationship between these components.

A storage client uses a vOSD-specific interface to initiate storage requests, such as the creation/removal of objects and I/O on their content. These requests are communicated to

the vOSD storage domain by the vOSD client-side driver using the communication channel. The job of the vOSD storage domain is to service these requests and provide mediated access to the physical storage devices.

The vOSD storage domain works as a *backend* for client vOSDs. It is a distributed service and is composed of one or more storage servers. Each storage server has multiple sub-components, described as follows:

- Storage virtualization. This component implements the support of multiple client vOSDs over shared physical storage and any conversions that might be required for client specific interfaces, such as converting data read from local disks into NFS read responses. This component also works as the back-end of the vOSD, in that any action performed on the vOSD is received by this component. These actions are checked for appropriate access restrictions and converted into requests for the storage management component.

- Storage management. This component facilitates the management of physical storage devices. In particular, it implements the allocation of physical storage corresponding to the objects in the vOSD and it implements any operations on them, including I/O.

- Access Control Module (ACM). This component enforces per-object access control and is a key element for implementing security, privacy and trust for the clients. The basis for access control are *labels* attached to clients and to objects. These labels behave as *capabilities*. Each object contains one or more labels, which are matched according to a policy with the label of the clients accessing the object. Labels associated with clients are *not* provided by the ACM itself, and are not part of the communication protocol between the vOSD storage domain and the storage client. Rather, *clients' labels are provided by an external trusted entity, the hypervisor*. This delegation greatly simplifies the storage domain architecture, since ACM need only implement enforcement, and not deal with label (capability) management of clients at all. Issues related with this management include capability generation, dissemination, enforcing expiration, and dealing with security aspects of the communication protocol,

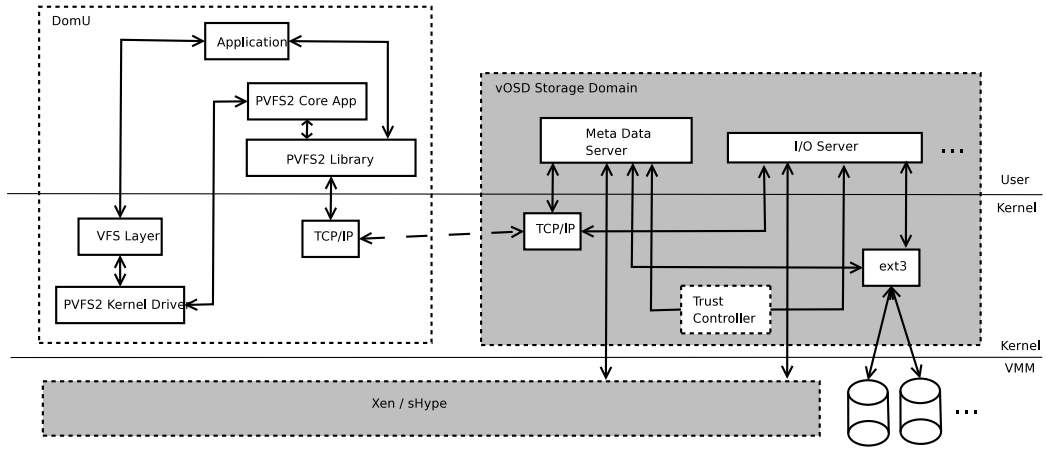


Figure 26: PVFS based object-oriented storage system.

such as spoofing and replay attacks [72]. In contrast, other storage systems, such as Lustre [21], implement this management of client capabilities as a part of the storage system itself.

Such access controls are in addition to any access controls implemented by guest virtual machines based on certain user credentials, such as user and group identifiers.

6.4 Implementation

Our prototype implementation of O2S2 architecture is based on the PVFS file-system [55, 90]. It runs on a platform virtualized with Xen [110]. Figure 26 shows the different components of this implementation.

A distributed file-system is chosen as a way to implement the prototype because such file-systems are commonly used in distributed environments to implement storage solutions that (1) enable resource consolidation on servers and (2) allow data sharing among multiple clients. Such sharing is enabled by the file-system's provision of a higher-level abstraction to clients. Our choice of the PVFS cluster file-system, rather than using NFS or Coda, is due to its ability to separate meta-data and data, and because it makes it possible to distribute data among multiple servers for performance. These properties also enable extensions that can provide differentiated storage. We specifically chose PVFS2 since it is mostly implemented in user space, which makes it easy to modify and debug, unlike, e.g., Lustre [21], which is implemented in the kernel. Further, PVFS2 is a freely available,

mature, and stable product.

6.4.1 PVFS Background

The client side vOSD is provided as a PVFS volume where objects are stored as PVFS files. The core of the PVFS file-system is implemented in user space. User-level applications can use a PVFS specific API and its client library to make PVFS system calls to access these files. These files can also be accessed by unmodified user space applications relying on the kernel's VFS layer to hide file system details. To enable this, the PVFS file-system provides a kernel driver that registers the file system with the kernel's VFS layer. This enables mounting the vOSD in the file hierarchy of the client. Any I/O in vOSD file space is directed to the PVFS kernel driver by the VFS layer. The kernel driver marshals the VFS request into a PVFS request and communicates it to a user-level application, called the *PVFS core*. This application works as a proxy to make PVFS system calls on behalf of the kernel driver. Similarly, all responses received by this application are communicated back to the driver, which converts it into appropriate VFS responses and hands it to the VFS layer.

Currently, the PVFS client component does not interface with the client kernel's page cache. Bypassing the page cache makes the file system design simple, since there is no client level consistency to maintain in a shared environment. However, this also means that every VFS request must be communicated to the storage domain. This reduces the performance for I/O operations with small block sizes. In contrast, PVFS performs well for large block sizes and large files.

The vOSD storage domain is implemented as multiple storage servers, each of which is a PVFS server. In our prototype implementation, all of these servers execute in the Dom0. Hence, the vOSD storage domain provides an example of a host-based iConnect realization, as described in Chapter 4.

A PVFS server is a user-level entity and is classified as either a meta-data server (MDS) or a I/O server. As the name suggests, a MDS manages meta-information about a PVFS file, such as attributes (length, last modification timestamp etc.) and access-control information

(*label* of the file, list of user ids allowed access etc.). Some of the meta-information depends on the type of file. For example, for a regular PVFS file, its meta-data contains information about the I/O servers that are in use for storing the data. PVFS also supports extended attributes, which can be user or system defined arbitrary key-value pairs. I/O servers are used to store actual data associated with a PVFS file.

PVFS servers use a combination of Berkeley DB [102] and files in the underlying file system – the former is used to store meta-data whereas data is stored in the latter. The PVFS filesystem might create multiple chunks for a PVFS file in order to parallelize access to data, each of which is a file stored at an I/O server.

Each PVFS server manages a storage space, which is a part of the local file system that it uses to store data. In our implementation, different storage spaces reside on different physical disk partitions. These disk partitions can be spread among multiple disks.

The communication between a PVFS client, i.e., a guest VM, and PVFS servers, i.e., the storage servers of the vOSD storage domain utilizes TCP/IP networking over virtual NICs provided by Xen.

6.4.2 Extensions to the PVFS-based Storage Domain

The vOSD storage domain that provides enhanced object-based storage is realized as an extension of the core PVFS implementation.

First, additional attributes, called *IOHints*, and access control labels, called *acm-labels*, are associated with a PVFS server's storage space. By associating a particular physical disk partition to a storage space, these IOHints and acm-labels also extend to the level of a physical disk partition. This is the lowest granularity of meta-data managed by the vOSD storage domain.

Second, with each PVFS file object stored in client's vOSD, two extended attributes are associated – *user.iohint* and *system.acmlabel*. The former is modifiable by the user, while the latter is only modifiable by the vOSD storage domain. We also extend the PVFS client library, and correspondingly the server-side PVFS implementation, to include a file-system call, called *extended create* or *ecreate* to create a file with specified extended attributes.

Access control is implemented by extending the PVFS server to include certain checks. These checks are included in the *prelude* part of every request serviced by the server. The type of access control implemented is Mandatory Access Control [66], where access rights of an object are non-transferable from one client to another without the intervention of the hypervisor and the vOSD storage domain. The ACM utilizes the information provided by the *secure hypervisor* (sHype) extension of Xen hypervisor [119] and, optionally, by the trust controller, described later.

Xen's sHype extension implements a repository of one static label per VM and one static label per physical resource, such as NIC and harddisk partition. Based on a matching policy, it also enforces mandatory access control – a VM can only access a physical resource if they both have the same label. These labels can be viewed as roles, hence this type of access control is an example of Role Based Access Control (RBAC) [66]. Currently, Xen supports bind time access control, i.e., labels are only matched at the time a guest VM is created. Also, the label associated with a VM does not change for the lifetime of a VM. This may change in the future, e.g., based on the identity of the physical platform on which the VM executes, a VM's label could change.

Since sHype does not define labels for anything other than VMs and physical resources, services like the vOSD storage domain must currently implement access control themselves. Toward this end, we use sHype as a repository for guest VM labels. The vOSD storage domain keeps its own labels for each server's and for each PVFS file object stored, as described earlier. Also, since the information sHype only maintains label information for clients specific to a physical platform, it implies that all storage servers of the vOSD storage domain need to coexist on the same platform. However, it is possible to distribute them among multiple machines by incorporating a distributed trust management solution, such as shamon [96] in the O2S2 architecture.

6.4.2.1 Trust Controller

A trust controller is an optional component that can be utilized with the storage domain to enhance its access control enforcement. The functionality of a trust controller is to maintain

a "trust" value for a guest VM. It utilizes one or more monitoring components that provide behavioral information about the guest VM. The "trust" value is a function of current behavioral information and of a trust policy. In our prototype, behavioral information is captured using a network monitoring component (*netmon*), which resides in the Dom0 kernel and essentially, extends the Xen virtual NIC backend by monitoring the network traffic to/from a guest VM.

Figure 27 shows the interaction between netmon and the trust controller. Netmon operates as follows. Based on a rule engine, it intercepts certain packets. Contents of these packets, such as headers of different protocol stacks and payload, generate the desired behavioral information. Further details about the netmon prototype along with performance analysis are described elsewhere, as part of the ProtectIT framework [83].

The current netmon prototype provides information pertaining to remote access to a guest VM, such as the number of open telnet and ssh connections, and the amounts of data transferred by these open connections. This information is exported to the trust controller using the */proc* interface. Additionally, netmon notifies the trust controller proactively when the information monitored by netmon changes. Since the trust controller is a part of the storage domain prototype in user space, kernel space netmon utilizes standard kernel-to-user space asynchronous signals for notification. The trust controller, then, accesses the network information via the /proc interface, and updates the "trust" value of the guest VM based on a specific trust model. These updates in "trust" value affect the overall access control for the guest VM, depending on the specific policy in use by the storage domain. An example trust model, along with example access control policies are described in detail in Section 6.5.1.

6.4.3 Discussion – Alternative Choices

Although our current prototype of the O2S2 architecture uses PVFS, the architecture itself is generic and can utilize alternative means to implement its components. Some of these alternatives are discussed in this section. Table 5 also summarizes these alternatives.

The client side virtual object-storage device (vOSD) is characterized by the interface

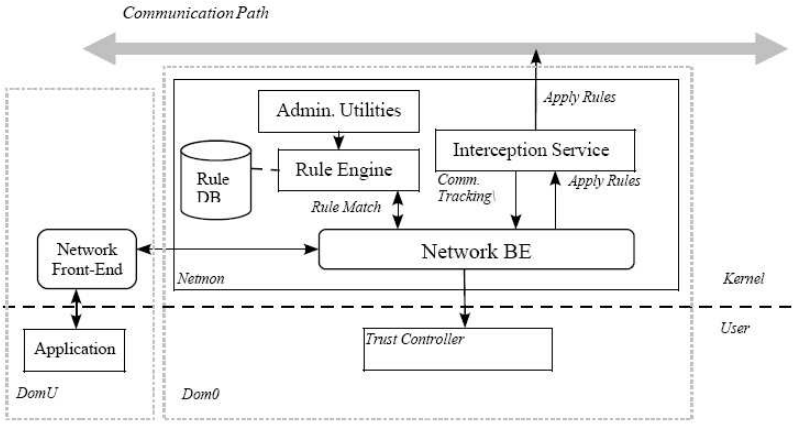


Figure 27: Interaction between trust controller and *Netmon*.

Table 5: Summary of design choices for various components of O2S2 architecture.

Component	Choices
vOSD interface to client	Filesystem (NFS, PVFS), T10
Communication Protocol between the vOSD storage domain and vOSD	Filesystem specific, T10 based
Storage Server	User process, Kernel service, Specialized VM
Objects for storage management	Files on local file-system, OSD objects

it provides to the client kernel. As an alternative to the file-system API, it could utilize a device access protocol, such as T10 [140], which is an enhancement of SCSI. In this case, the device driver sets up the virtual device as part of the SCSI device stack as a SCSI initiator. A shim file system layer [137] is required to present a file-system interface on top of this virtual device, which then interfaces to the VFS layer in the kernel. Alternatively, as is the case with PVFS and other distributed file-systems, this device can also be directly accessed by user space applications via libraries.

The client vOSD driver formats storage requests as messages based on a specific protocol and forwards them to the storage domain. For example, a vOSD based on NFS can use NFS-specific messages to communicate with the storage domain. However, it is also possible for a vOSD to provide an interface to the client that is entirely different from the one used to access the storage domain. For example, a vOSD providing a PVFS interface to the storage client could convert PVFS-specific messages to T10 commands that can then be sent to a storage domain which works as the OSD target [62]. Another example of such a protocol

conversion is when the storage domain performs this conversion prior to performing actual storage virtualization, management and access control tasks [74].

Since the vOSD storage domain is a distributed service, there are many ways to implement each storage server: as a user-level application (e.g., our current prototype), as a kernel-level service (e.g., Lustre), or as a runtime inside a dedicated and specialized VM of its own. Many commercial SAN and NAS implementations also fall into the first two categories. Although we are not aware of an example of a storage domain comprised of multiple specialized VMs, an approach similar to that of Libra [47] is also feasible for implementing a storage server. IBM's Tiburon project [37] uses a similar approach.

Another related issue for implementing the vOSD storage domain concerns the malleability of the enhanced functionalities being exported (i.e., whether there is a predefined set and/or whether a client must choose among them, or whether the set can be defined by the client). Since these functionalities are domain specific, e.g., they may differ for multimedia vs. file storage, there are multiple approaches for providing an interface for defining these functionalities. Examples include a domain specific language [7] and arbitrary binaries executing inside isolated containers, such as processes and VMs. In this work, we focus on a fixed set of functionalities offered by the vOSD storage domain – the dynamic extension of storage domains by a client is part of our future work.

The job of the storage virtualization component is to map a vOSD object access request to a set of objects managed by the storage management component. The storage management component manages these objects with the help of the underlying platform's storage services. For example, our current PVFS based prototype and many other cluster file-system servers use local file systems, such as ext3, of the machines on which they run to store these objects as files. If the underlying platform has OSDs attached to it, these objects could be provided by the device itself. In this case, the storage management component runs on the OSD that houses the particular object being accessed. In a similar fashion, ACM could be located on the device itself.

6.5 Functionalities

This section describes various functionalities of the vOSD storage domain implementation and demonstrates how these functionalities provide enhanced vOSDs to storage clients.

6.5.1 Object-based Access Control

The vOSD storage domain implements per-object, multi-layer, role-based access control (RBAC). RBAC is based on *labels* associated with clients, physical storage devices, and individual objects stored in those devices.

On each request, a storage server first determines the client's label. Currently, each client is a VM physically located on the local machine with IPs in a private subnet. By using the association of a client's IP address with its VM id, the storage server obtains the client's label from Xen/sHype. This label is then cached for the lifetime of the VM. Next, this label is matched against labels of the storage space being managed by the server. In case of a mismatch, the request is denied. Otherwise, if a request does not pertain to a specific object in the storage space, such as a request to obtain the PVFS configuration from the storage domain's MDS, the access control returns success, and the request is allowed to continue.

If a request pertains to a specific object, labels of that object (stored as system.acmlabel extended attributes) are matched with the client's label. In case of a mismatch, the request is denied, otherwise is allowed to continue.

For each PVFS file created by the client in vOSD, all of the objects stored in the storage space of a server inherit the client's label. For example, for a file create operation by a client with label *WebSurfing*, the meta-data object and one or more data-objects, created on MDS and I/O servers respectively, contain system.acmlabel attribute set as *WebSurfing*. It is possible to append more labels to an object from a privileged client (such as Dom0) – hence an object can be shared among multiple clients. Alternatively, a "group" type label can be utilized for sharing purpose, where the hypervisor maintains the association of a client to a group, and the individual object is labeled with the "group" label. The latter approach is currently part of our future work.

A client's label need not be statically defined. It can be dynamically computed based on a function of the initial static label assigned by the VMM and the "trust" value of the client, as determined by the trust controller. This dynamic label is equivalent to a dynamic role assignment for the client – hence the access control implemented by the storage domain based on dynamic role assignment can be viewed as a special form of RBAC, termed dynamic RBAC. Dynamic RBAC provides more flexibility than static RBAC, which only utilizes static labels.

The trust controller computes the "trust" values of the guest VM based on a simple trust model, which defines floating values for "trust" in the range $(0.0, 1.0]$. These "trust" values are based on the number of open telnet and ssh connections with the VM as the server, computed according to the following formula:

$$\text{"trust"} = 1/(1 + \text{number of active ssh connections} + \text{number of active telnet connections}) \quad (1)$$

The information related to the number of open connections is provided by netmon, as described earlier.

Based on this floating "trust" value and the client VM's static label (label_{static}), the storage domain can use different policies to generate client VM's dynamic label ($\text{label}_{dynamic}$). Two such policies and their application scenarios are described next.

Policy 1

if "trust" $== 1.0$ **then**
  $\text{label}_{dynamic} = \text{label}_{static}$
else if "trust" < 1.0 **then**
  $\text{label}_{dynamic} = \text{NULL}$
end if

A NULL label by default denies any access. Policy 1 implies that when a VM has any open ssh or telnet connections, it cannot access any object in the vOSD.

Policy 2 defines dynamic labels for different "trust" values. A dynamic label is constructed by appending access restrictions to the static label. Access restrictions are represented as an AND of NOTs $((!T_1)\&(!T_2)\&\cdots\&(!T_n))$, where each $T_i, i \in \{1, \cdots, n\}$ is a specific access type. Examples of these access types used in policy 2 are:

- DW – data write,
- DR – data read,
- MDRW – meta-data read write.

For example, a dynamic label $\text{label}_{dynamic} = \text{label}_{static\ \&\ (!DW)\ \&\ (!DR)}$ evaluates to NULL for access types of data write and data read, but evaluates to label_{static} for meta-data read write. This dynamically evaluated label is then matched against the object's label for access control purposes.

Policy 2

if "trust" == 1.0 **then**
  $\text{label}_{dynamic} = \text{label}_{static}$
else if "trust" $\geq$ 0.50 **then**
  $\text{label}_{dynamic} = \text{label}_{static\ \&\ (!DW)}$
else if "trust" $\geq$ 0.33 **then**
  $\text{label}_{dynamic} = \text{label}_{static\ \&\ (!DW)\ \&\ (!DR)}$
else if "trust" $\geq$ 0.25 **then**
  $\text{label}_{dynamic} = \text{label}_{static\ \&\ (!DW)\ \&\ (!DR)\ \&\ (!MDRW)}$
end if

Object-based access control enables the efficient sharing of physical devices, since the granularity of access control can be per-object rather than targeting larger disk partitions. Hence, objects from multiple vOSDs can be stored on the same physical disk, which reduces fragmentation and increases utilization. Second, objects from multiple vOSDs can be shared among multiple clients – by assigning multiple labels to this object pertaining to multiple clients. Third, a storage domain level access control can be a part of a multi-layer access control solution [106]. For example, a client may impose further role-based access control for multiple users, such as in SELinux [29].

6.5.2 Semantic Aggregation of Multiple Storage Devices

In a heterogeneous storage environment, e.g., a home/personal environment, the vOSD storage domain collectively utilizes the ensemble to storage devices to store objects from various vOSDs. An object is stored on the storage space of a specific storage server, based on its attributes (e.g., *user.iohint* and type), and *IOHints* of the storage server. For example,

an object with user.iohint attribute set to 'X' can only be stored at server(s) whose storage space contains 'X' (or *generic*) as a IOHint. These labels can be chosen based on any policy, e.g., physical device specific characteristics. Such semantic aggregation of multiple storage devices based on objects' properties enables *differentiated storage* for a vOSD. Here, costs associated with deciding which storage server to use are one time and are paid at the time when an object is created. Afterwards, an object's I/O performance depends on the I/O performance of the specific storage server(s), which itself depends on the share of resources (such as disk bandwidth and CPU) associated with these server(s).

Differentiated storage provides multiple benefits, including easier data management and performance isolation among multiple clients and multiple types of applications in a client. Performance isolation is provided by storing isolated objects on different servers using different disks for storage space. An example is storing metadata on a MDS using a faster flash based disk, while storing sequential data on an I/O server using a high capacity SCSI disk. In this fashion, the vOSD storage domain can minimize the performance impact of extensive meta-data I/O performed by a VM, e.g., searching for a particular file, on a VM that performs streaming data I/O, e.g., watching a movie. Easier management results from the fact that data can be stored on a device based on its utility, as suggested earlier in Section 6.2.

6.6 Experimental Evaluation

Experimental evaluation of the prototype O2S2 implementation is carried out on a dual-core, 3GHz, 64-bit x86 CPU based server class machine with 1GB RAM and 160GB SATA, 7200 RPM hard disk with 8MB cache. The hypervisor used is Xen version 3.0.4 with sHype enabled. The policy used by sHype is a simple type enforcement policy. The privileged VM, Dom0, is assigned 512MB RAM and exclusive access to one physical processor. The second processor is shared among different guest VMs. For our experiments, each guest VM is configured with 128MB RAM. Both Dom0 and guest VMs run a para-virtualized Linux kernel based on version 2.6.16.33. The vOSD storage domain, which resides in Dom0's user space, and vOSDs it provides to a client are based on PVFS version 2.6.3.

6.6.1 Microbenchmarks

These experiments measure the basic costs of implementing enhanced functionality in the storage domain, namely object-level access control and differentiated storage. For the latter, we also provide a quantitative evaluation of the potential benefits in terms of the performance isolation it provides.

6.6.1.1 Access Control Module

Performance For analyzing the costs associated with ACM, we execute various PVFS system calls on a vOSD from a guest VM, with and without ACM present in the vOSD storage domain. The storage domain runs two storage servers – one MDS and one I/O server, both sharing the same disk for their storage space. The application executing these system calls directly uses PVFS's client libraries, without having to go through the kernel's VFS layer.

Figure 28 depicts the normalized latency comparison for some of these system calls with and without ACM, using the latter as the base. The figure also includes the total number of access control checks performed for each system call, followed by its name on the x-axis in the following format: (#Server ACM checks + #Object ACM checks). Since ACM caches a client's labels from Xen/sHype, the cost of a Server ACM check involves label matching only. The cost of an Object ACM check requires accessing its extended attribute (*system.acmlabel*) from PVFS's storage, followed by label matching. We find the cost of a Server ACM check and an Object ACM check to be $\sim 8\mu s$ and $\sim 62\mu s$, respectively. Also, the latency of read (write) system call depend on the block size being read (written). Since the cost of ACM checks is fixed per system call, the normalized latency with ACM checks for read and write system calls will change based on the block size. The measurements presented in Figure 28 are the costs for 32MB size blocks. Since the application makes single blocking I/O requests, I/O throughput can be computed as (block size/latency of system call). For this block size, we find the read and write throughput to be 170.5 MB/s and 40.81 MB/s, respectively, with ACM, as compared to 174.81 MB/s and 40.98 MB/s, respectively, without ACM. These results demonstrate that the ACM component minimally

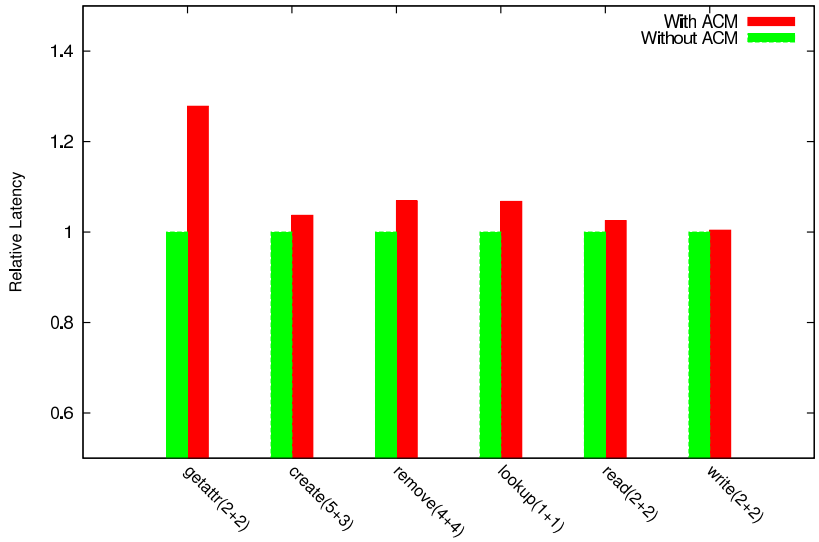


Figure 28: Performance comparison of PVFS system calls with- and without-ACM in the vOSD storage domain.

impacts the performance of PVFS system calls, both in terms of latency and throughput.

Scalability We use write throughput to demonstrate the impact of ACM component on a storage domain's scalability. Figure 29 shows the relative performance profile of a storage domain with increasing number of VMs, using single VM measurements as the base case. Based on memory size, the maximum number of VMs configured with 128MB RAM handled by our test platform is 3 (theoretically, the limit is 4, given that there is no memory overcommitment; however, the VMM uses a certain amount of memory itself, resulting in a limit of 3). Here, single VM numbers for both, with- and without-ACM, cases are normalized to one. In case of more than one VM with ACM (without ACM), cumulative performance of all VMs relative to the single VM with ACM (without ACM) is shown, along with individual components. A near-identical performance profile demonstrates that adding ACM functionality does not impact the scalability of a storage domain.

Dynamic Access Control As stated earlier, the ACM module offers dynamic role-based access control (RBAC) functionality. This is demonstrated by dynamically changing the network related behavior of a guest VM and by showing its effect on three storage workloads

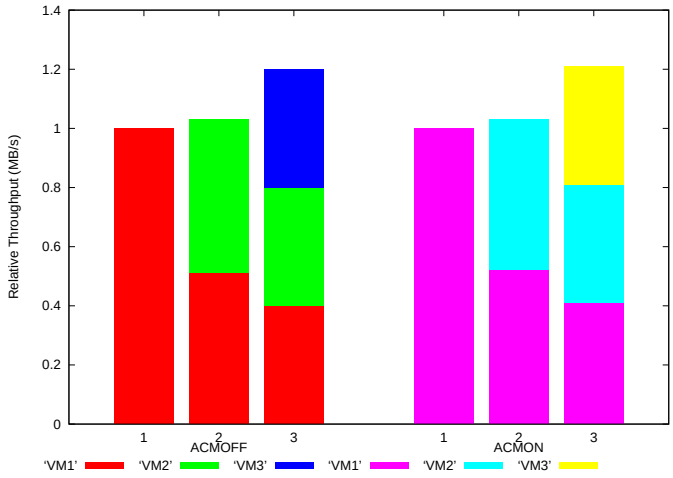


Figure 29: Scalability of the vOSD storage domain.

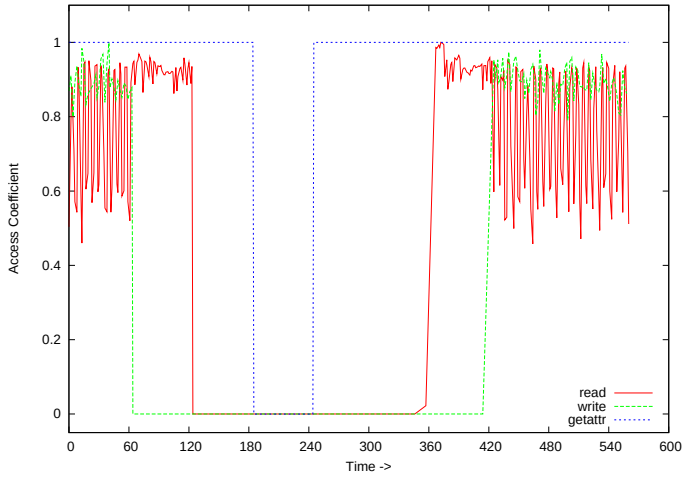


Figure 30: Effect of dynamic RBAC on different workloads.

running in the guest VM. The first workload continuously writes to a file; the second workload continuously reads from a file; and the third workload continuously reads attributes of a file (i.e., performs meta-data reads). These workloads are identified as *write*, *read* and *getattr*, respectively. Figure 30 shows the time line on X-axis and behavior of workloads on the Y-axis in terms of *Access Coefficient*. A value of access coefficient greater than 0.0 indicates successful access, while a value of 0.0 indicates an access failure. For data read and write workloads, the access coefficient is computed as the ratio of instantaneous throughput and maximum throughput achieved. For meta-data read write workload, success of access results in access coefficient value of 1.0, while failure of access, in 0.0.

A script executing in the Dom0 incrementally opens four ssh connections to the guest VM, and then incrementally drops them. The script opens connections one and two at 60 seconds and 120 seconds, respectively, and opens connections three and four at 180 seconds. These connections are dropped at 60 seconds intervals starting at 240 seconds. These dynamic changes in number of network connections affect the "trust" value of the guest VM, computed based on the Equation 1. The storage domain uses policy 2 to define dynamic roles for the guest VM, which affects its access to the objects stored in the vOSD.

As demonstrated by the results, the data write workload can successfully function except in time period $(60, 420)$ seconds. This is due to the fact that the "trust" value of the guest VM in this time period remains < 1.0, which prohibits write access to the objects for the VM. Similarly, the data read workload and meta-data read workload can successfully function expect in time periods $(120, 360)$ and $(180, 240)$, in which the "trust" values are < 0.50 and < 0.25, respectively. These results show the ability of the storage domain to dynamically adapt the access control imposed on a guest VM with changes in its "trust" value, as perceived by the trust controller.

6.6.1.2 Differentiated Storage

For differentiated storage, we compare the latency of two PVFS system calls – *ecreate* and pre-existing *create*, both performed from a user-level application inside a VM, directly using the PVFS libraries. In this experiment the vOSD storage domain consists of three storage servers – one MDS, one I/O server with label *mobile*, and other I/O server with label *fixed*. Based on the value of the *user.iohint* attribute specified with the *ecreate* call, the storage domain chooses one of the I/O servers for storing the object data. With the *create* call, the client chooses an I/O server in a round-robin fashion. Table 6 shows the latency of these system calls, as measured from the client VM. The difference between the latencies is due to extra processing performed in the MDS to match the attribute with *IOHints* of various I/O servers, and due to additional I/O in the MDS to store the extended attribute, the latency of which is based on the disk being used for the storage space at the MDS. For this experiment, all storage servers share the same SCSI disk. We expect the latency to be

Table 6: Cost of implementing differentiated storage.

Operation	Latency (ms)
create	19.504
ecreate	22.393

lower for different kinds of storage, such as a flash disk, which provides better performance for short, random, reads of extended attributes [131].

In order to show the performance benefits of differentiated storage, we run the vOSD storage domain with two servers, one MDS and one I/O server. There are two competing VMs performing different kinds of I/O activities. One VM, VM1, is continuously creating new extended attributes for an object, while the other VM, VM2, is doing writes to the same object. We plot the performance of VM2 executing simultaneously with VM1 in two scenarios – one, where both MDS and I/O server share the same SCSI disk for their storage space, termed 'perturbation', and two, where MDS uses a ramdisk for its storage space while the I/O server uses the SCSI disk, termed 'perturbation with differentiated-storage'. The throughput of VM2 without the presence of VM1 is used as the base line, labeled as 'no perturbation', and the performance of VM2 in scenarios described above is plotted relative to this base line performance. Figure 31 shows VM2's write throughput for different block sizes.

These results demonstrate that VM1 can substantially degrade VM2's I/O performance, since both MDS and I/O server must share the disk I/O path, along with the CPU. However, with differentiated storage, the vOSD storage domain provides much better performance isolation. Performance impact with differentiated storage emanates from the fact that the MDS and the I/O server share the same CPU to serve VM1 and VM2, respectively. We expect the performance with differentiated storage to be even closer to the base line in future multicore machines, where different storage servers can be provided with separate physical CPUs.

6.6.2 IOzone Benchmark

To understand the performance implications of the ACM component on application-level workloads, we evaluate the I/O performance of our prototype storage domain using the

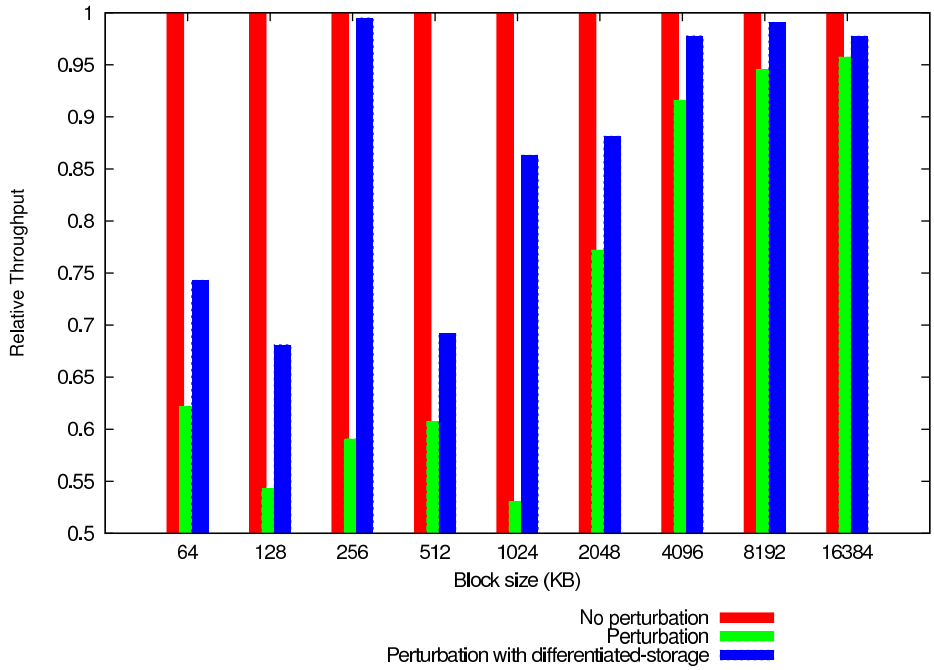


Figure 31: Performance isolation in vOSD storage domain.

IOzone benchmark [18]. The IOzone benchmark measures file I/O performance for many file-system operations, such as read, write, and mmaped I/O. This benchmark is executed on a client VM, and accesses the vOSD via kernel's VFS interface. The goal of these experiments is to demonstrate that (1) using ACM component does not significantly impact the I/O performance of the vOSD storage domain, and (2) the vOSD storage domain can provide significant performance benefits to a client VM, along with enhanced functionality, by utilizing Dom0's computational resources, which is an option that is not available to a block-based virtual disk (VBD) based on a physical disk partition.

For IOzone's write throughput experiments, we also enable an additional parameter for storage servers, called *NoDataSync* (NDS). This option allows storage servers to buffer writes before they are flushed to the disk. Without this option, every write is flushed to the disk. Using this option allows the vOSD storage domain to provide much better performance by reducing the latency of each write operation. The tradeoff is that the storage domain may loose data in the event of a server failover. However, using redundancy with this option enabled could provide similar performance benefits, at a reduced risk of failure.

As mentioned earlier, the PVFS file system does not utilize a client's page cache, and

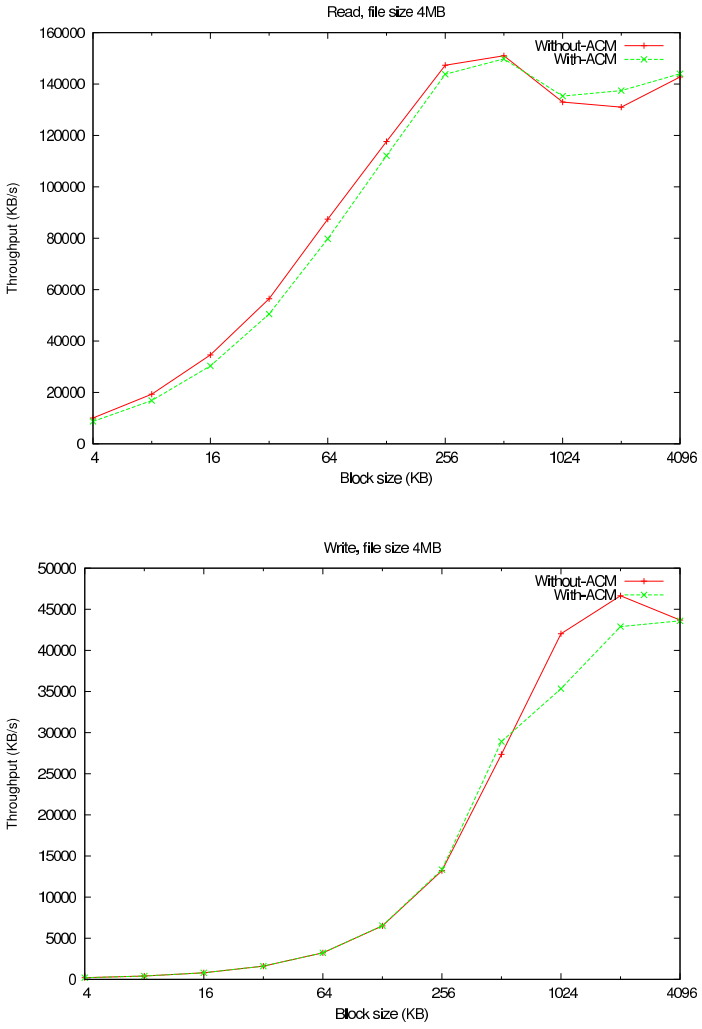


Figure 32: IOzone small I/O performance.

hence the I/O throughput of the vOSD storage domain for small block sizes is limited – anywhere from .8% upto 14% of that of a VBD. Keeping this limitation in mind, for small read and writes, we only demonstrate the comparative performance of the vOSD storage domain with- and without-ACM. Figure 32 depicts read and write throughput with varying block sizes for a file of size 4MB. Since ACM imposes a fixed cost on each I/O operation, with increasing block sizes, overall cost for access checks decreases for a fixed file size due to a decrease in total number of I/O operations being performed. The *relative performance*, measured as small quantity/larger quantity for each block size, varies within $(86.5\%, 99.3)$ and $(84\%, 100\%)$ of each other, for read and write respectively. These results demonstrate that the ACM component minimally impacts the I/O performance of the vOSD storage domain.

For large I/O, we include results for two file sizes, 128MB and 512MB, respectively. We also include the results for VBD. These results are shown in Figures 33 and 34. For a 128MB file, the client VM's page cache is no longer effective. Hence the performance of read for VBD is around 45MB/s. However, vOSD continues to enjoy the benefits of the vOSD storage domain's page cache, and hence can provide read throughput of upto 150MB/s. For writes, the performance of vOSD without NDS trails the performance of VBD, since every write must go to disk in both cases – however, vOSD's path to physical disk is longer than VBD. However, vOSD with NDS uses asynchronous write in the storage domain, and hence can provide write throughput of $\sim$ 120MB/s, upto $\sim 2X$ of that of VBD.

For a 512MB file, both the client VM's and the vOSD storage domain's page caches are no longer effective, but larger memory in the storage domain still enables better performance for vOSD for large block sizes, since virtual network I/O is much faster than hard disk access. For writes, vOSD without NDS is slower than VBD, for the same reason as described above. However, vOSD with NDS can overlap asynchronous writes with disk I/O, and hence can provide $\sim 1.5X$ performance gain over VBD.

Similar to the previous case of small file size, there is minimal performance impact of ACM component on the vOSD storage domain for large file sizes. The relative performance of read varies within $(98.5\%, 99.6\%)$ for a 128MB file, and within $(88.1\%, 99.9\%)$ for a

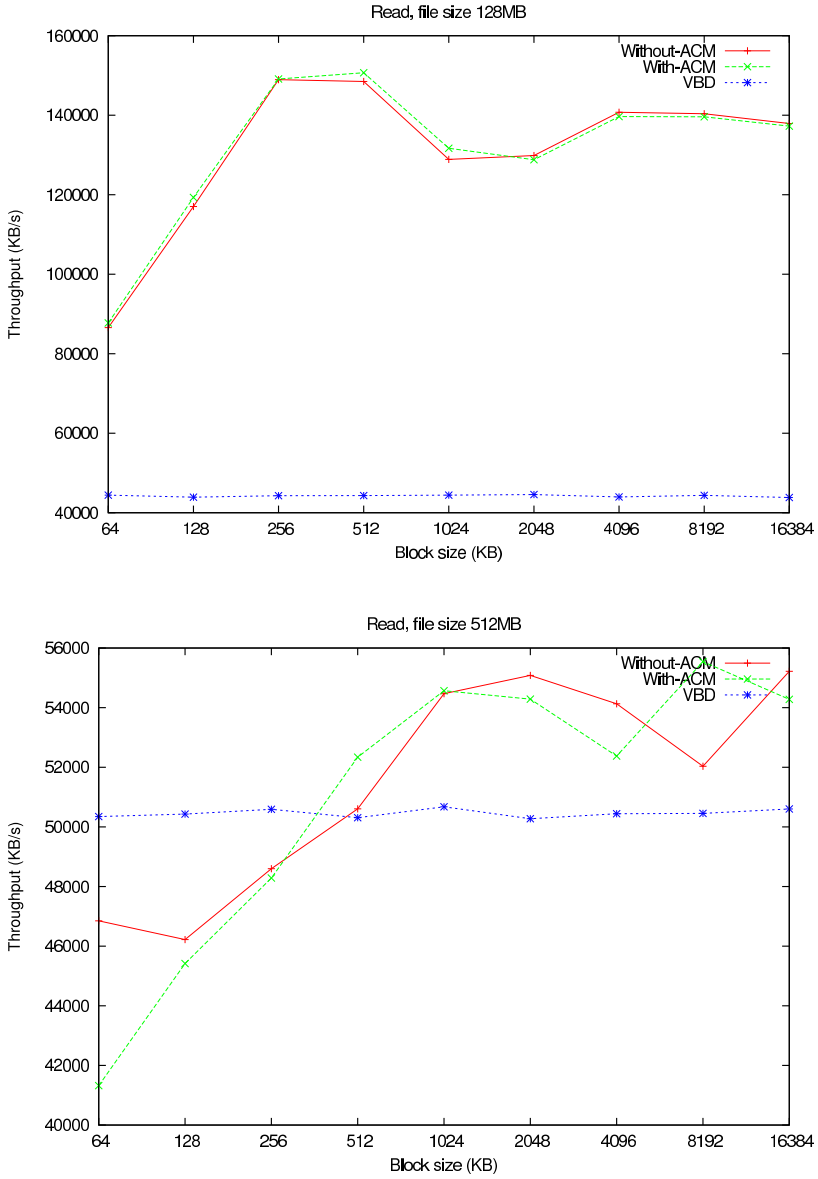


Figure 33: IOzone large read performance.

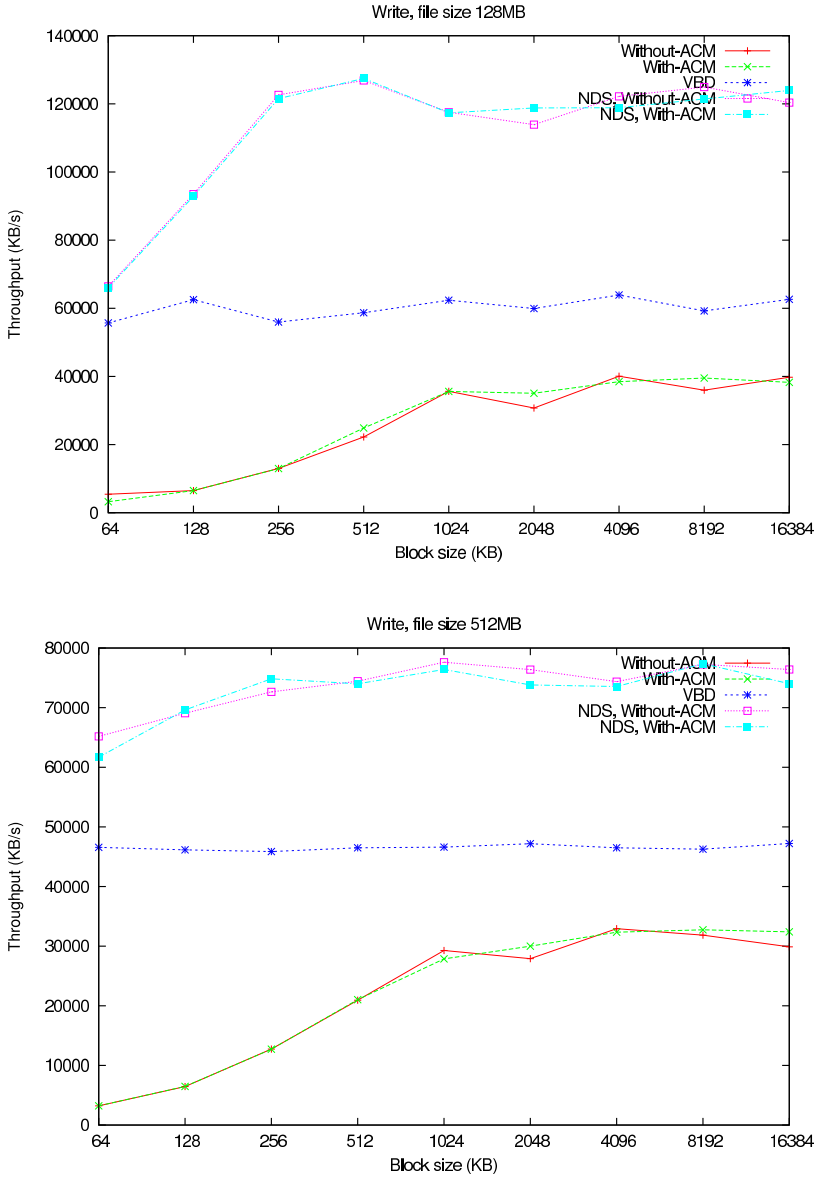


Figure 34: IOzone large write performance.

512MB file. Similarly, for writes, it varies within $(89\%, 99.9\%)$ for a 128MB file, and within $(92\%, 100\%)$ for a 512MB file.

The IOzone benchmark results demonstrate that the enhanced access control functionalities of the storage domain can be implemented with minimal performance overhead for the guest VMs. In particular, the throughput for read and write operations degrades only minimally when utilizing the ACM component. Also, an object based storage domain makes it possible to better exploit the resources available at the storage domain as compared to a block based storage virtualization solution.

6.7 Conclusions and Future Work

This chapter presents an object-based storage system architecture, and a PVFS file-system based prototype implementation, called the *vOSD storage domain*. The storage virtualization service, enabled by the vOSD storage domain, provides virtual object-storage devices (vOSDs) to VMs in a virtualized environment. An object-based interface not only allows for *efficient sharing of physical devices*, it also enables *dynamic, role-based, access control* and *usability based performance isolation* in a heterogeneous storage environment. Performance results demonstrate that our storage domain implementation provides enhanced functionalities without adversely affecting the performance and scalability of the storage service. Further, by efficiently utilizing Dom0's resources, the storage domain can also provide certain performance benefits to client VMs, such as the use of its page cache as a storage cache.

In the future, the PVFS based interface for the client can be replaced with a standardized T10 interface [140]. This permits the vOSD to be entirely de-coupled from the specific vOSD storage domain implementation. Translation between the T10 interface and the underlying storage domain's storage access mechanisms will be implemented in the vOSD storage domain itself. To this effect, Xen's virtual SCSI frontend [130] can be enhanced to make it compatible with T10, and similarly, the virtual SCSI backend can be merged with the storage domain. Additionally, it is possible to utilize a different file-system backend, LWFS [101], which promises efficient I/O in large scale systems and more flexibility for

implementing per-object properties, such as checkpointing.

As part of future work, the current vOSD storage domain implementation can be extended to make it distributed, such that various storage servers can be located on different physical machines. This will require integration with a distributed trust management infrastructure, such as shamon [96]. Also, extending the vOSD storage domain with logging infrastructure for SLA auditing will enhance its security and trust related properties, as discussed in Section 6.2.2.1.

CHAPTER VII

RELATED WORK

7.1 Extensible, Self-Virtualized I/O

Modern computer systems perform a variety of tasks, not all of which are suitable for execution on general purpose processors. A platform consisting of both general purpose and specialized processing capability can provide the high performance as required by specific applications. A prototype of such a platform is envisioned by Hady et al [73] where a CPU and a NP are utilized in unison. Due to the limitations of the PCI interconnect, a special interconnect is designed that provides better bandwidth and latency for CPU-NP communication. More recently, multiple heterogeneous cores have been placed on the same chip [33]. These cores can share resources at a much lower level than shared memory (such as L2 cache), thus greatly improving both bandwidth and latency of inter-core communication. Our work uses a similar heterogeneous platform, consisting of Xeon CPUs and an IXP2400 NP communicating via a PCI interconnect. For this platform, performance advantages are demonstrated for a device-centric realization of SV-IO. This is in comparison to other solutions that use general purpose cores for network packet processing and other device-near tasks [115, 53].

The SV-IO abstraction bears resemblance to the virtual channel processor abstraction proposed by McAuley et al [95], although the intended use for virtual channel processors is to provide virtual devices for some system-level functionality, such as iSCSI, rather than guests having to run their own iSCSI module which in turn communicates to the virtual network interface. VIFs provided by SV-IO can be similarly enhanced by added functionality, as demonstrated in Chapter 4.

In order to improve network performance for end user applications, multiple configurable and programmable network interfaces have been designed [147, 109]. These interfaces could also be used to implement a device-centric SV-NIC. Another network device that implements

this functionality for the zSeries virtualized environment is the OSA network interface [26]. This interface uses general purpose PowerPC cores for the purpose, in contrast to the NP cores used by our SV-NIC. We believe that using specialized network processing cores provides performance benefits for domain specific processing, thereby allowing more efficient and scalable SV-IO implementation. Furthermore, these virtual interfaces can be efficiently enhanced to provide additional functionality, such as packet filtering and protocol offloading.

Our SV-NIC uses VMM-bypass in order to provide direct, multiplexed, yet isolated, network access to guest domains via VIFs. The philosophy is similar to U-Net [135] and VMMC [63], where network interfaces are provided to user space with OS-bypass. A guest domain can easily provide the VIF to user space applications, hence SV-NIC trivially incorporates these solutions. Similarly, new generation InfiniBand devices [91] offer functionality that is akin to the ethernet-based SV-NIC, by providing virtual channels that can be directly used by guest domains. However, these virtual channels are less flexible than our SV-NIC in that no further processing can be performed on data at the device level.

Although NPs have generally been used standalone for carrying out network packet processing in the fast path with minimal host involvement, previous work has also used them in a collaborative manner with hosts, to extend host capabilities, such as for fast packet filtering [50]. We use the NP in a similar fashion to implement the self-virtualized network interface. Application-specific code deployment on NPs and other specialized cores has been the subject of substantial prior work [117, 70, 152].

7.2 Sidecore

Substantial prior research has addressed benefits of utilizing dedicated cores, both in heterogeneous [87, 73] and homogeneous [115] multicore systems. Self-virtualized devices [111] provide I/O virtualization to guest VMs by utilizing the processing power of cores on the I/O device itself. In a similar manner, driver domains for device virtualization [110] utilize cores associated with them to provide I/O virtualization to guest VMs. The sidecore approach presented in this work utilizes dedicated host core(s) for system virtualization tasks. Particularly, we advocate the partitioning of the VMM's functionalities and utilizing

dedicated core(s) to implement a subset of them. A similar approach is used in operating systems, where processor partitioning is used for network processing [114, 53].

Computation spreading [56] attempts to run similar code fragments of different threads on the same core and dissimilar code fragments of the same thread on different cores. Another approach is to run hardware exceptions on a different hardware thread (or core) instead of running it on the same thread (core) [153]. While these solutions are targeted for better utilization of the micro-architecture resources such as caches, branch predictors, instruction pipeline etc., our solution is targeted at improving VMM performance and scalability for large scale many-core systems.

Intel's McRT (many-core run time) [118] in sequestered mode uses dedicated cores to run application services in non-virtualized systems. This approach requires major modifications in the application to utilize the parallel cores. This is in contrast to the sidecore approach, which requires only minor modifications to the guest VM's kernel and is aimed at improving the overall system performance.

7.3 iConnect and Logical Devices

Past research has made multiple attempts at providing semantically enhanced devices/interconnects in order to provide useful functionality to applications. For example, semantic-disks associate filesystem level information with the disk drive [123] to provide better performance and functionality to operating systems. Similarly, application-specific handlers can be used at the enhanced network devices [70]. The iConnect approach makes a similar argument, albeit in a virtualized system, where guest VMs' interactions are semantically enhanced by iConnect. In particular, for enhancing a guest VM's I/O, the iConnect abstraction provides what amount to logical devices. This approach is similar to Xen's 'soft' devices [139]. However, our implementation also utilizes the underlying platform's capabilities for supporting efficient I/O in virtualized systems, e.g., self-virtualized devices [111, 91], and it could take advantage of other computational resources like accelerators [128].

Entities implementing the virtualization service (such as a storage domain presented

in Chapter 6, a driver domain providing network virtualization in a virtualized environment [110], or a VMedia runtime [113] for multimedia device virtualization) can use their computational resources to implement additional functionalities/properties for logical devices. This is not necessarily the case when these are directly supported by the underlying physical hardware itself. Examples include image manipulation in VMedia, additional computations on data in active storage, either in storage servers [108], or in the hardware itself [116], and TCP segment offload in virtual NICs [97].

Elevating the abstraction to include semantic information enables efficient utilization and sharing, as demonstrated by significant past research in distributed file systems [52, 28]. Furthermore, standardized higher level interfaces improve both the ease of implementation and the adoption of such solutions. The VMGL [89] approach virtualizes a video card to provide hardware-based 3D acceleration to guest VMs, and uses the OpenGL abstraction as the interface. The MVAPICH2-ivc library [77] encapsulates the shared-memory based inter-VM communication on a single physical host, and provides the MPI abstraction for HPC applications running inside guest VMs. These approaches are similar to ours – the VMedia framework uses the Video4Linux [40] interface and the O2S2 instance evaluated in this work uses the PVFS [55] interface.

Aggregating multiple devices to provide richer services has been studied along several dimensions. Superimposed projection [61] discusses fundamental issues arising when using multiple projectors to produce a single high-resolution image. The Princeton scalable display wall project [136] also discusses algorithms to solve alignment, color balancing, and other problems arising in a distributed environment. The Lyra system [151] studies timing services that can be provided to multimedia applications, for achieving better quality of service. Such services can be harnessed in multimedia scheduling in a virtualized environment to provide QoS guarantees to guest VMs and to schedule fine-grained captures in frame aggregation (Section 5.4.4). Storage services have also utilized aggregation of multiple storage devices to increase the storage capacity and/or to provide other properties such as reliability and fault tolerance [59]. These storage services, specifically secondary storage caches [133], can utilize timing services to provide additional properties, such as freshness

of content.

The semantic information considered in specific instances of logical devices described in this work is explicitly shared by guest VMs with the virtualized platform. This is not necessary though – it is possible to implicitly infer limited amounts of information by monitoring a guest VM's behavior. For example, a VMM can infer when memory pages are added and removed from the guest OS page cache [79]. This implicit inference allows building VMM-level services such as a working-set size estimator and better secondary cache management. Similarly, it is possible to infer application level communication topology in a VM-based grid environment [127]. This topology information can be used to adapt the environment itself, e.g., via VM migration or via communication overlay adaptation, to provide performance benefits to applications running inside guest VMs.

7.3.1 Multimedia Virtualization

The Irisnet project [71] enables multimedia sharing via filtering on distributed multimedia sensors to deliver customized content. Feeds from several remote webcams connected to the Internet are used to compose useful content and services built on top. MSODA [150] proposes a multimedia service overlay among virtual machines for media service access and composition. VMedia framework focuses on providing multimedia services to virtual machines via higher level 'logical' devices, while services are implemented in the Service VM. The Indiva middleware [103] also provides a higher level, file system abstraction, for composing distributed multimedia content.

7.3.2 Object-based Virtualized Storage

The O2S2 architecture shares its object based design with many distributed file-systems, such as Lustre [21], Panasas [143] and Ceph [141]. While the main focus of these file-systems has been high performance and scalability, it is possible to extend them with enhanced per-object functionality and properties explored as part of this work. A similar approach is taken by Piernas et al [108], where they extend Lustre with active storage functionality.

Storage systems utilize various data properties as hints for storage management, such as data encoding, fault model, timing model [42] and frequency of access [81]. These properties

can also influence other properties. For example, frequency of access can indicate whether certain data should be stored compressed [54], or decide reliability guarantees provided to it [146]. These properties can be similarly incorporated as hints in the O2S2 architecture at an object level.

Previous research in security management for network attached storage, both at an object-level [72] and block-level [46], utilizes un-forgeable cryptographic capabilities issued by storage servers to enforce access control. In contrast, we use a multi-layer approach, where capabilities external to the storage system, *labels* provided by the VMM, are utilized to enforce access control. These capabilities are used in a manner that is oblivious to storage clients. Our approach is similar to using external hardware components, such as Trusted Platform Modules (TPM) [39], to store and provide capabilities about an potentially untrusted execution entity.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

8.1 Thesis

This research contributes efficient methods for system virtualization, specifically for I/O, for multi-core and future many-core systems. Its solutions efficiently use the parallelism provided by multiple computational cores and judiciously utilize memory and I/O bandwidth to provide improved performance to guest virtual machines. They also leverage semantic information available from guests to provide them with enhanced functionalities. These enhancements use the paradigm of virtualization services, examples of which are services that generalize or specialize certain platform capabilities, ranging to services that offer entirely new, typically software-realized functions.

The approach and solutions advocated in this research have had substantial impact, demonstrated not only by several publications resulting from this research, but also by recent architectural enhancements in Intel's new platform support for I/O virtualization [16]. In these enhancements, I/O queues with associated controls on DMA mappings realize the virtual interfaces (VIFs) advocated and implemented in our research, thereby providing in hardware some of the functionality this work implements for self-virtualized devices. These emerging trends also help validate the basic premises of this thesis, that new virtualization methods are needed to cope with the increasing mismatch of computational capabilities vs. memory and I/O bandwidths in multi- and many-core machines. To this end, the virtualization service based approach presented in this work provides new methods to re-architect the I/O virtualization sub-system, and provides examples of services that provide novel and enhanced functionality. Extensive performance evaluation of these virtualization services validate our thesis, that virtualization services provide scalable performance in multi-core platforms. These performance benefits result from better resource utilization based on re-architecting the solution, and from exploitation of semantic information.

The remainder of this chapter summarizes this dissertation's main contributions and offers comments on future research.

8.2 Research Contributions

Advocating a service-oriented approach to system virtualization, we demonstrate that virtualization services, coupled with re-architecting core virtualization components can improve the performance and scalability of multi- and many-core platforms. Focusing on I/O, we present the abstraction of Self-Virtualized I/O as a basis for implementing virtualization services. Implementations of multiple instances of virtualization services serve to demonstrate the viability and advantages of the concept, in terms of improved performance or reliability and by providing entirely new functionality to guest VMs. Several examples of such functionality are listed below.

8.2.1 Software Artifacts

Software artifacts, i.e., realizations of virtualization services created as part of this research include the following:

- SV-NIC: a software framework for providing a network virtualization service based on the IXP2XXX network processor platform, e.g., the ENP2611 [6]. The major software entities of this framework, all available under GNU public license, are: (1) a host side management kernel driver for Dom0, (2) a patch for the Xen HV for interrupt virtualization, (3) management and network virtualization software for NP-based platforms, and (4) a kernel driver for guest VMs to access virtual NICs. This software framework provides a realization of device-based SV-IO, the SV-NIC. A network virtualization service implementation by the SV-NIC demonstrates that by (1) utilizing the cores judiciously in the heterogeneous platform and (2) re-architecting the various components of the I/O virtualization solution to optimize the usage of I/O sub-system, it is possible to obtain better scalability and performance. The SV-NIC also enables logical devices via including semantic information about the VIFs, such as QoS information 4.2.2. Hence, the virtualization service provided by the SV-NIC enables

enhanced functionality for the virtualized platform, with no cost imposed on host components.

- Sidecore: a patch for the Xen HV to componentize one or more host CPU cores as sidecores. This also includes software entities that utilize a sidecore for interrupt virtualization for SV-NIC and low-latency VM-VMM communication in VT-enabled systems. The sidecore approach demonstrates the benefits of componentization and functional partitioning approaches for homogeneous host cores to further enhance the performance of virtualization services.

- SV-NIC supported RVBD: an enhancement to SV-NIC in order to provide storage service to guests. The functionality is similar to NBD, except that RVBD can be utilized without any networking support from guest VM. This demonstrates that the storage virtualization service provided by the SV-NIC can enable enhanced functionality with better performance, while reducing the complexity of the guest VMs. These benefits are derived by exploiting the semantic information associated with the service.

- VMedia framework: an extensible multi-media device virtualization service. Two concrete software entities are: (1) a kernel driver for guest VM for virtual media device, and (2) a userspace runtime for the Dom0 that can aggregate multiple multimedia devices, such as cameras. The framework allows semantic sharing of multimedia data and performs computations in the Dom0 to support various attributes of virtual media devices The multimedia virtualization service demonstrates novel methods for sharing multimedia devices based on semantic information, which enables better performance and scalability for this service in a multi-core system. This service also demonstrates that enhanced functionality can be implemented for guest VMs with low or no cost to guest VMs themselves.

- Security and Trust enhanced Object Store: an extension to the PVFS file system that provides fine grain, per-object, mandatory access control based on the labels of client guest VMs, which are managed by the underlying virtualized platform. This access control also incorporates, and can in turn influence, the "trust" of a guest

VM, which is managed by a trust-controller. All the functionality of this service is implemented in the Dom0, without any participation from a guest VM, and outside the core communication protocols of the file-system. In this way, this service is impervious to security attacks that otherwise affect traditional storage services, where communication protocols themselves include necessary security mechanisms, such as cryptographic capabilities. This storage virtualization service demonstrates that by exploiting semantic information and higher-level API, virtualization services can provide not only better performance and scalability, but also novel functionality for guest VMs and for the platform with minimal performance impact.

8.2.2 Evaluation Results

Instances of virtualization services presented in this dissertation demonstrate qualitative and quantitative benefits of the service based approach for system virtualization, specifically for I/O. The SV-NIC prototype presented in Chapter 2 shows that a device-centric approach to network virtualization can better exploit the computational resources of a heterogeneous multi-core system, providing substantial improvements in performance (upto $\sim 2X$ better throughput and $\sim 50\%$ less latency) and scalability. Similarly, by judiciously partitioning the cores among different functionalities of a virtualization service, the performance and scalability of the SV-NIC can be further improved, as demonstrated in Chapter 3.

Chapters 4, 5 and 6 provide instances of virtualization services that exploit semantic information to provide novel functionality and/or better performance for guest VMs, at little or no extra cost to these VMs. In particular, the VMedia prototype presented in Chapter 5 shows upto an order of magnitude or more performance improvement for multimedia device sharing performed with semantic information as compared to a sharing solution without semantic information. The object-based storage service prototype evaluated in Chapter 6 provides enhanced security and trust with negligible reduction in performance. This object-based solution can also provide upto $2X$ better I/O performance as compared to a block-based solution for large data files.

8.3 Future Research Directions

The notion of virtualization service has longer term importance because it can be used both to enhance existing platform capabilities and to support entirely new functionality. This can be exploited to create what appear to guest VMs as homogeneous, fully interoperable platforms, at some costs in performance [99]. It can also be used to create virtual execution platforms with functionalities not yet present in hardware or better realized with software solutions. For example, by implementing additional logical functionality, virtualization services can create virtual devices that exhibit characteristics not supported or supportable by underlying physical devices. A specific instance of such functionality is multi-device aggregation, where multiple remote disks are utilized to provide a higher 'quality' disk. Quality dimensions may include improved survivability due to the spatially distributed nature of physical disks, improved reliability due to the use of replication, or improved performance exploiting concurrent device access. More interestingly, virtualization services provide a framework for exploring entirely novel device functionalities and properties. For example, virtualization services can exploit the security and trust information provided by the underlying virtualization infrastructure, as demonstrated by the O2S2 storage service prototype in Chapter 6. The implementation of these services use request tagging and tracking to enable per-request device and content access controls. Beyond such direct effects on device use, virtualization services can also monitor and analyze the ways in which guest VMs use devices, including collecting certain behavioral or semantic information about guest VMs. Examples include QoS monitoring [45], trust monitoring [83], monitoring to effect network related adaptations [127], and monitoring to improve memory management [79].

The specific virtualization services presented in this work provide a fixed set of functionalities to guest VMs via logical devices. Any extension or modification to these functionalities must be performed by the service provider itself. In order to facilitate logical devices with guest-specific functionality, the current approach must be supplemented with an extension framework, which should allow a guest VM to provide modules that can be dynamically deployed with the I/O virtualization component of the service to form guest-specific logical devices. Such a dynamic service composition framework will provide more

flexibility to guest VMs, since they can selectively choose to offload functionality to the virtualization service itself. Depending on the availability of resources at the virtualized platform providing the service, this may result in performance benefits for the guest VM.

Dynamically deployable functional modules raise expressibility and security concerns similar to those present in other extensible environments, such as kernel modules in current OSes and extensions in the SPIN microkernel [49]. At one extreme, these modules could be comprised of arbitrary binary code, and the virtualization service can use a hardware protection mechanism, such as paging and segmentation [60], for fault isolation. For example, the extension can execute inside a VM, while I/O virtualization task executes inside another VM. At the other extreme, fault isolation can be based on software mechanisms, such as a domain specific type-safe language. Alternatively, a hybrid mechanism to ensure both safety and expressibility can be utilized [69]. For example, guest-specific extensions for the VMedia framework can be composed in ECL, a rich subset of C [64]. Such a domain-specific language based hybrid approach may provide a reasonable tradeoff between the cost imposed by the runtime and the expressibility of guest specific extensions.

An interesting attribute of virtualization services is that they can be used to functionally partition the hypervisor and I/O functions present in multi-core platforms. Such partitioning can be used to improve the performance of such platforms, as shown in this thesis with the side-core implementation of hypervisor calls. An interesting future direction of research is to consider dynamic core partitions based on current guest VM behavior and available platform resources. This would entail replacing our current static mapping of hypervisor components to multi-core resources with dynamic mappings, possible based on current platform state (e.g., whether or not certain cores are in certain states, including idle vs. active states). Although the number of cores available in a system are expected to increase, a virtualization service should be enhanced to handle dynamic changes in architectural resources, in particular to the number and types of computational cores available to it. In this manner, the utility of the overall system can be optimized with changing environment. For example, if the guest VMs in a system do not fully utilize computational cores, these cores could be utilized by the virtualization services to provide enhanced functionality to

these VMs. However, if the demand for computational cores from guest VMs increases, the system should be able to reposition some of these computational resources away from the virtualization service. Such dynamic repositioning of resources should not disable a virtualization service – it should continue functioning, albeit with possible performance degradation.

As demonstrated by the object-based storage virtualization service, there are multiple benefits of integrating security and trust functionalities provided by the underlying platform. In order to attain these benefits for virtualization services in a distributed environment comprising of multiple physical machines, it is imperative that these security and trust management solutions themselves be distributed. Ongoing and future work in this area will address the mechanisms and policies for trust management in a distributed environment [96]. By integrating virtualization services with distributed, virtualization aware, trust management solutions, we can provide secure virtualization services for the enterprise. Another aspect of this integration involves building better trust models that accurately reflect the "trust" properties of a VM. These models will utilize the behavioral information provided by various monitoring components, such as XenAccess [105] and netmon [83]. These trust models will enable virtualization services to implement better and more meaningful dynamic access control policies.

REFERENCES

[1] "Alacritech SES1800 Series iSCSI Scalable Network Accelerators for Volume Servers." http://www.alacritech.com/Products/Storage/SES1800.aspx, accessed February, 2007.

[2] "Amazon Elastic Compute Cloud." aws.amazon.com/ec2, accessed May, 2007.

[3] "Amazon Simple Storage Service." aws.amazon.com/s3, accessed May, 2007.

[4] "AMD I/O Virtualization Technology Specification." http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs%/34434.pdf, accessed October, 2006.

[5] "CamE." http://directory.fsf.org/camE.html, accessed April, 2007.

[6] "ENP-2611 Data Sheet." http://www.radisys.com/files/ENP-2611_07-1236-05_0504_datasheet.pdf, accessed October, 2005.

[7] "EVPath." http://www.cc.gatech.edu/systems/projects/EVPath/, accessed April, 2007.

[8] "First Look: Maxtor's NAS Offers Simple Solution." PCWorld Magazine article, published March 15, 2005. http://www.pcworld.com/article/id,120063-page,1/article.html, accessed May, 2007.

[9] "Global Environment for Network Innovations." http://www.geni.net, accessed October, 2007.

[10] "Hypertransport interconnect." http://www.hypertransport.org/, accessed February, 2006.

[11] "IBM eserver xSeries ServeRAID Technology." ftp://ftp.software.ibm.com/pc/pccbbs/pc_servers_pdf/raidwppr.pdf, accessed October, 2005.

[12] "IBM Systems Virtualization." http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay%.pdf, accessed February, 2006. IBM Corporation, Version 2 Release 1 (2005).

[13] "imlib." http://freshmeat.net/projects/imlib/, accessed April, 2007.

[14] "Intel 21555 Non-transparent PCI-to-PCI Bridge." http://www.intel.com/design/bridge/21555.htm, accessed May, 2005.

[15] "Intel Pentium D Processor Specification." http://www.intel.com/products/processor/pentium_D/index.htm, accessed February, 2006.

[16] "Intel Virtualization Technology for Connectivity." http://softwarecommunity.intel.com/isn/downloads/virtualization/pdfs/20%137_LAD_VTc_Tech_Brief_r04.pdf, accessed October, 2007.

[17] "Intel Virtualization Technology Specification for the IA-32 Intel Architecture." ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf, accessed February, 2006.

[18] "IOzone File System Benchmark." www.iozone.org, accessed May, 2007.

[19] "Iperf." http://dast.nlanr.net/projects/Iperf, accessed October, 2005.

[20] "Linux-VServer." http://linux-vserver.org/Paper, accessed May, 2007.

[21] "Lustre: A Scalable, High-Performance File System." http://www.lustre.org/docs/whitepaper.pdf, accessed May, 2007.

[22] "Network Block Device." http://nbd.sourceforge.net, accessed April, 2006.

[23] "NFE-i8000 Network Acceleration Card." Information available from http://netronome.com/web/guest/products/acceleration_cards, accessed October, 2007.

[24] "ObjectStone." http://www.haifa.il.ibm.com/projects/storage/objectstore/objectstone.ht%ml, accessed May, 2007.

[25] "OpenVZ." http://openvz.org, accessed May, 2007.

[26] "OSA-Express for IBM eserver zSeries and S/390." www.ibm.com/servers/eserver/zseries/library/specsheets/pdf/g2219110.pdf%, accessed October, 2005.

[27] "PCI Express IO Virtualization and IO Sharing Specification." http://www.pcisig.com/members/downloads/specifications/pciexpress/speci%fication/draft/IOV_spec_draft_0.3-051013.pdf, accessed October, 2006.

[28] "RFC 3530: Network File System version 4 Protocol." www.ietf.org/rfc/rfc3530.txt, accessed May, 2007.

[29] "Security-Enhanced Linux." http://www.nsa.gov/selinux/, accessed May 2007.

[30] "SNORT Intrusion Detection System." www.snort.org, accessed May, 2006.

[31] "Tcpdump/Libpcap." http://www.tcpdump.org/, accessed October, 2005.

[32] "Teraflops Research Chip." http://techresearch.intel.com/articles/Tera-Scale/1449.htm, accessed October, 2007.

[33] "The Cell Architecture." http://www.research.ibm.com/cell/, accessed February, 2006.

[34] "The Linux Logical Volume Manager." http://www.redhat.com/magazine/009jul05/features/lvm2/, accessed May, 2007.

[35] "The Open Archives Initiative Protocol for Metadata Harvesting Guidelines : Provenance." http://www.openarchives.org/OAI/2.0/guidelines-provenance.htm, accessed October, 2007.

[36] "The VMWare ESX Server." http://www.vmware.com/products/esx/, accessed June, 2005.

[37] "Tiburon." http://www.almaden.ibm.com/storagesystems/projects/tiburon/, accessed May, 2007.

[38] "Tickless Idle." http://www.lesswatts.org/projects/tickless/, accessed October, 2007.

[39] "Trusted Platform Module (TPM) Specification." https://www.trustedcomputinggroup.org/specs/TPM, accessed February, 2007.

[40] "Video4linux resources." http://www.exploits.org/v4l, accessed February, 2007.

[41] "XenAccess Library." http://xenaccess.sourceforge.net/, accessed May, 2007.

[42] Abd-El-Malek, M., William V. Courtright, I., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J., "Ursa Minor: Versatile Cluster-Based Storage," in *Proc. of FAST*, 2005.

[43] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B., and Wiegert, J., "Intel Virtualization Technology for Directed I/O," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[44] Adams, K. and Agesen, O., "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proc. of ASPLOS*, 2006.

[45] Agarwala, S., Chen, Y., Milojicic, D., and Schwan, K., "QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems," in *The 3rd IEEE International Conference on Autonomic Computing (ICAC 2006)*, 2006.

[46] Aguilera, M. K., Ji, M., Lillibridge, M., MacCormick, J., Oertli, E., Andersen, D., Burrows, M., Mann, T., and Thekkath, C. A., "Block-Level Security for Network-Attached Disks," in *Proc. of FAST*, 2003.

[47] Ammons, G., Appavoo, J., Butrico, M., Silva, D. D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Hensbergen, E. V., and Wisniewski, R. W., "Libra: a Library Operating System for a JVM in a Virtualized Execution Environment," in *Proc. of VEE*, 2007.

[48] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the Art of Virtualization," in *Proc. of SOSP*, 2003.

[49] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility, Safety and Performance in the SPIN Operating System," in *Proc. of SOSP*, 1995.

[50] Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., and Portokalidis, G., "FFPF: Fairly Fast Packet Filters," in *Proc. of OSDI*, 2004.

[51] Bova, T. and Krivoruchka, T., "Reliable UDP Protocol." Available as IETF draft from http://www3.ietf.org/proceedings/99mar/I-D/draft-ietf-sigtran-reliable-%udp-00.txt, accessed October, 2005.

[52] Braam, P. J., "The Coda Distributed File System," *Linux Journal*, vol. 50, June 1998.

[53] Brecht, T., Janakiraman, J., Lynn, B., Saletore, V., and Turner, Y., "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[54] Burrows, M., Jerian, C., Lampson, B., and Mann, T., "On-Line Data Compression in a Log-Structured File System," in *Proc. of ASPLOS*, 1992.

[55] Carns, P. H., Ligon, W. B., Ross, R. B., and Thakur, R., "PVFS: A Parallel File System For Linux Clusters," in *Proc. of the 4th Annual Linux Showcase and Conference*, (Atlanta, GA), October 2000.

[56] Chakraborty, K., Wells, P. M., and Sohi, G. S., "Computation Spreading: Employing Hardware Migration to Specialize CMP cores on-the-fly," in *Proc. of ASPLOS*, 2006.

[57] Chambliss, D. D., Alvarez, G. A., Pandey, P., Jadav, D., Xu, J., Menon, R., and Lee, T. P., "Performance Virtualization for Large-Scale Storage Systems," in *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[58] Chandra, R., Zeldovich, N., Sapuntzakis, C., and Lam, M. S., "The Collective: A Cache-Based System Management Architecture," in *Proc. of NSDI*, 2005.

[59] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A., "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, 1994.

[60] cker Chiueh, T., Venkitachalam, G., and Pradhan, P., "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions," in *Proc. of SOSP*, 1999.

[61] Damera-Venkata, N. and Chang, N. L., "Realizing Super-Resolution with Superimposed Projection," in *Proc. of IEEE International Workshop on Projector-Camera Systems*, 2007.

[62] Devulapalli, A., Dalessandro, D., Wyckoff, P., Ali, N., and Sadayappan, P., "Integrating Parallel File Systems with Object-based Storage Devices," in *Proc. of Supercomputing*, 2007.

[63] Dubnicki, C., Bilas, A., Li, K., and Philbin, J., "Design and Implementation of Virtual Memory-Mapped Communication on Myrinet," in *Proc. of the International Parallel Processing Symposium*, 1997.

[64] Eisenhauer, G., Bustamante, F., and Schwan, K., "Event Services for High Performance Computing," in *Proc. of High Performance Distributed Computing*, 2000.

[65] Factor, M., Meth, K., Naor, D., Rodeh, O., and Satran, J., "Object Storage: The Future Building Block for Storage Systems," in *Proc. of the Second International IEEE Symposium on Emergence of Globally Distributed Data*, 2005.

[66] Ferraiolo, D. and Kuhn, R., "Role-Based Access Control," in *Proc. of 15th NIST-NCSC National Computer Security Conference*, 1992.

[67] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M., "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," in *Proc. of OSDI*, 1999.

[68] Ganev, I., *A Pliable Hybrid Architecture for Run-time Kernel Adaptation*. PhD thesis, College of Computing, Georgia Institute of Technology, 2007.

[69] Ganev, I., Eisenhauer, G., and Schwan, K., "Kernel Plugins: When a VM is Too Much," in *VM'04: Proc. of the 3rd conference on Virtual Machine Research And Technology Symposium*, 2004.

[70] Gavrilovska, A., Mackenzie, K., Schwan, K., and McDonald, A., "Stream Handlers: Application-Specific Message Services on Attached Network Processors," in *Proc. of the Symposium on High Performance Interconnects (HotI)*, 2002.

[71] Gibbons, P. B., Karp, B., Ke, Y., Nath, S., and Seshan, S., "IrisNet: An Architecture for a World-Wide Sensor Web," *IEEE Pervasive Computing*, October-December 2003.

[72] Gibson, G. A., Nagle, D. P., Amiri, K., Chang, F. W., Feinberg, E., Lee, H. G. C., Ozceri, B., Riedel, E., and Rochberg, D., "A Case for Network-Attached Secure Disks," Tech. Rep. CMU-CS-96-142, CS, Carnegie Mellon University, 1996.

[73] Hady, F., Bock, T., Cabot, M., Chu, J., Meinechke, J., Oliver, K., and Talarek, W., "Platform Level Support for High Throughput Edge Applications: The Twin Cities Prototype," *IEEE Network*, vol. 17, July/August 2003.

[74] Hildebrand, D. and Honeyman, P., "Direct-pNFS: Scalable, Transparent, and Versatile Access to Parallel File Systems," in *Proc. of HPDC*, 2007.

[75] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S., "Practical Taint-Based Protection Using Demand Emulation," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[76] Huang, L., Peng, G., and cker Chiueh, T., "Multi-Dimensional Storage Virtualization," in *Proc. of SIGMETRICS*, 2004.

[77] Huang, W., Koop, M. J., Gao, Q., and Panda, D. K., "Virtual Machine Aware Communication Libraries for High Performance Computing," in *Proc. of Supercomputing*, 2007.

[78] Intel Corporation, "Intel IXP2400 Network Processor: Hardware Reference Manual," October 2003.

[79] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, 2006.

[80] Kim, C., Burger, D., and Keckler, S. W., "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *Proc. of ASPLOS*, 2002.

[81] Kohl, J. T., Staelin, C., and Stonebraker, M., "HighLight: Using a Log-Structured File System for Tertiary Storage Management," in *Proc. of the USENIX Winter 1993 Technical Conference*, 1993.

[82] Kong, J., Ganev, I., Schwan, K., and Widener, P., "CameraCast: Flexible Access to Remote Video Sensors," in *Proc. of MMCN*, 2007.

[83] Kong, J., Schwan, K., Lee, M., and Ahamed, M., "ProtectIT: Trusted Distributed Services Operating on Sensitive Data," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2008.

[84] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Silva, D. D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V., "K42: Building a Complete Operating System," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[85] Kumar, R., *Holistic Design for Multi-core Architectures*. PhD thesis, Department of Computer Science and Engineering, UCSD, 2006.

[86] Kumar, S., Agarwala, S., and Schwan, K., "Netbus: A Transparent Mechanism for Remote Device Access in Virtualized Systems," Tech. Rep. GIT-CERCS-07-08, CERCS, Georgia Tech, 2007. `http://www.cercs.gatech.edu/tech-reports/tr2007/git-cercs-07-08.pdf`, accessed October, 2007.

[87] Kumar, S., Gavrilovska, A., Schwan, K., and Sundaragopalan, S., "C-CORE: Using Communication Cores for High Performance Network Services," in *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA)*, 2005.

[88] Kumar, S., Raj, H., Schwan, K., and Ganev, I., "Re-architecting VMMs for Multicore Systems: The Sidecore Approach," in *Proc. of Workshop on Interaction between Operating System and Computer Architecture (WIOSCA)*, 2007.

[89] Lagar-Cavilla, H. A., Tolia, N., Satyanarayanan, M., and de Lara, E., "VMM-Independent Graphics Acceleration," in *Proc. of VEE*, 2007.

[90] Latham, R., Miller, N., Ross, R., and Carns, P., "A Next Generation Parallel File System for Linux Clusters," *Linux-World*, pp. 56–59, January 2004.

[91] Liu, J., Huang, W., Abali, B., and Panda, D. K., "High Performance VMM-Bypass I/O in Virtual Machines," in *Proc. of USENIX ATC*, 2006.

[92] Ma, X. and Reddy, A. N., "MVSS: An Active Storage Architecture," *IEEE Transactions On Parallel And Distributed Systems*, vol. 14, September 2003.

[93] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, 2002.

[94] Marty, M. R., Bingham, J. D., Hill, M. D., Hu, A. J., Martin, M. M., and Wood, D. A., "Improving Multiple-CMP Systems Using Token Coherence," in *Proc. of HPCA*, 2005.

[95] McAuley, D. and Neugebauer, R., "A Case for Virtual Channel Processors," in *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, 2003.

[96] McCune, J., Berger, S., Caceres, R., Jaeger, T., and Sailer, R., "Shamon: A Systems Approach to Distributed Trust," in *Proc. of the Annual Computer Security Applications Conference*, 2006.

[97] Menon, A., Cox, A. L., and Zwaenepoel, W., "Optimizing Network Virtualization in Xen," in *Proc. of USENIX ATC*, 2006.

[98] Menon, A., Santos, J. R., Turner, Y., Janakiraman, J., and Zwaenepoel, W., "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," in *Proc. of VEE*, 2005.

[99] Nathuji, R. and Schwan, K., "VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems," in *Proc. of SOSP*, 2007.

[100] Niranjan, R., Gavrilovska, A., and Schwan, K., "Towards IQ-Appliances: Quality-awareness in Information Virtualization," Tech. Rep. GIT-CERCS-07-06, CERCS, Georgia Institute of Technology, 2007.

[101] Oldfield, R. A., Maccabe, A. B., Arunagiri, S., Kordenbrock, T., Riesen, R., Ward, L., and Widener, P., "Lightweight I/O for Scientific Applications," in *Proc. of Cluster Computing*, 2006.

[102] Olson, M. A., Bostic, K., and Seltzer, M., "Berkeley DB," in *Proc. of USENIX ATC*, 1999.

[103] Ooi, W. T., Pletcher, P., and Rowe, L. A., "Indiva: A Middleware for Managing Distributed Media Environment," in *Proc. of MMCN*, 2004.

[104] Padala, P., Zhu, X., Wang, Z., Singhal, S., and Shin, K. G., " Performance Evaluation of Virtualization Technologies for Server Consolidation," Tech. Rep. HPL-2007-59, Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto, 2007.

[105] Payne, B. D., Carbone, M., and Lee, W., "Secure and Flexible Monitoring of Virtual Machines," in *Proc. of the Annual Computer Security Applications Conference*, 2007.

[106] Payne, B. D., Sailer, R., Caceres, R., Perez, R., and Lee, W., "A Layered Approach to Simplified Access Control in Virtualized Systems," *ACM SIGOPS Operating Systems Review*, vol. 41, July 2007.

[107] Petrini, F., Kerbyson, D., and Pakin, S., "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *Proc. of Supercomputing*, 2003.

[108] Piernas, J., Nieplocha, J., and Felix, E. J., "Evaluation of Active Storage Strategies for the Lustre Parallel File System," in *Proc. of Supercomputing*, 2007.

[109] Pratt, I. and Fraser, K., "Arsenic: A User Accessible Gigabit Network Interface," in *Proc. of INFOCOM*, 2001.

[110] Pratt, I., Fraser, K., Hand, S., Limpack, C., Warfield, A., Magenheimer, D., Nakajima, J., and Mallick, A., "Xen 3.0 and the Art of Virtualization," in *Proc. of the Ottawa Linux Symposium*, 2005.

[111] Raj, H. and Schwan, K., "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices," in *Proc. of HPDC*, 2007.

[112] Raj, H., Seshasayee, B., and Schwan, K., "VMedia: Enhanced Multimedia Services in Virtualized Systems," Tech. Rep. GIT-CERCS-07-19, CERCS, Georgia Institute of Technology, 2007. `http://www.cercs.gatech.edu/tech-reports/tr2007/git-cercs-07-19.pdf`, accessed July, 2007.

[113] Raj, H., Seshasayee, B., and Schwan, K., "VMedia: Enhanced Multimedia Services in Virtualized Systems," in *Proc. of MMCN*, 2007.

[114] Rangarajan, M., Bohra, A., Banerjee, K., Carrera, E. V., Bianchini, R., Iftode, L., and Zwaenepoel, W., "TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance," Tech. Rep. DCS-TR-481, Department of Computer Science, Rutgers University, 2002.

[115] Regnier, G., Minturn, D., McAlpine, G., Saletore, V. A., and Foong, A., "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," *IEEE Micro*, vol. 24, no. 1, pp. 24–31, 2004.

[116] Riedel, E. and Gibson, G., "Active Disks - Remote Execution for Network-Attached Storage," Tech. Rep. CMU-CS-97-198, CS, Carnegie Mellon University, 1997. `www.pdl.cmu.edu/PDL-FTP/NASD/CMU-CS-97-198.pdf`, accessed October, 2007.

[117] Rosu, M., Schwan, K., and Fujimoto, R., "Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture," in *Proc. of Cluster Computing*, 1998.

[118] Saha, B., Adl-Tabatabai, A.-R., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J., "Enabling Scalability and Performance in a Large Scale CMP Environment," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2007.

[119] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J., and van Doorn, L., "Building a MAC-based Security Architecture for the Xen Opensource Hypervisor," in *Proc. of the Annual Computer Security Applications Conference*, 2005.

[120] Salim, J. H., Olsson, R., and Kuznetsov, A., "Beyond Softnet," in *Proc. of 5th USENIX Annual Linux Showcase and Conference*, 2001.

[121] Schmuck, F. and Haskin, R., "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proc. of FAST*, 2002.

[122] Singleton, L., Nathuji, R., and Schwan, K., "Flash on Disk for Low-power Multimedia Computing," in *Proc. of MMCN*, 2007.

[123] Sivathanu, M., Prabhakaran, V., Popovici, F., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Semantically-Smart Disk Systems," in *Proc. of FAST*, 2003.

[124] Sprott, D. and Wilkes, L., "Understanding Service-Oriented Architecture," *Microsoft Architect Journal*, vol. 1, January 2004. `http://msdn2.microsoft.com/en-us/arcjournal/aa480021.aspx`, accessed July, 2007.

[125] Subbiah, A. and Blough, D., "An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments," in *Proc. of the Workshop on Storage Security and Survivability*, 2005. `http://users.ece.gatech.edu/~dblough/research/papers/storagess05.pdf`, accessed May, 2007.

[126] Sugerman, J., Venkitachalam, G., and Lim, B.-H., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proc. of USENIX ATC*, 2001.

[127] Sundararaj, A. I., Gupta, A., and Dinda, P. A., "Increasing Application Performance in Virtual Environments through Run-time Inference and Adaptation," in *Proc. of HPDC*, 2005.

[128] Tarditi, D., Puri, S., and Oglesby, J., "Accelerator: using data parallelism to program GPUs for general-purpose uses," in *Proc. of ASPLOS*, 2006.

[129] Thekkath, C. A. and Levy, H. M., "Limits to low-latency communication on high-speed networks," *ACM Trans. Comput. Syst.*, vol. 11, no. 2, 1993.

[130] Tomonori, F., "Xen scsifront/back Drivers." Presented at Fall Xen summit 2006, available from `http://xen.xensource.com/files/summit_3/xen-scsi-slides.pdf`, accessed May, 2007.

[131] Tuma, W., "Comparison of Drive Technologies for High-Transaction Databases." `www.storagesearch.com/soliddata-art2-comparisons.pdf`, accessed May, 2007.

[132] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U., "Towards Scalable Multiprocessor Virtual Machines," in *Proc. of the Virtual Machine Research and Technology Symposium*, 2004.

[133] Vazhkudai, S. S., Ma, X., Freeh, V. W., Strickland, J. W., Tammineedi, N., and Scott, S. L., "FreeLoader: Scavenging Desktop Storage Resources for Scientific Data," in *Proc. of Supercomputing*, 2005.

[134] Viswanath, R., Ahamed, M., and Schwan, K., "Harnessing Non-dedicated Wide-area Clusters for On-demand Computing," in *Proc. of the IEEE International Conference on Cluster Computing*, 2005.

[135] von Eicken, T., Basu, A., Buch, V., and Vogels, W., "U-Net: a user-level network interface for parallel and distributed computing," in *Proc. of SOSP*, 1995.

[136] Wallace, G., Anshus, O. J., Bi, P., Chen, H., Chen, Y., Clark, D., Cook, P., Finkelstein, A., Funkhouser, T., Gupta, A., Hibbs, M., Li, K., Liu, Z., Samanta, R., Sukthankar, R., and Troyanskaya, O., "Tools and Applications for Large-Scale Display Walls," *IEEE Computer Graphics and Applications*, vol. 25, July 2005.

[137] Wang, F., Brandt, S. A., Miller, E. L., and Long, D. D., "OBFS: A File System for Object-Based Storage Devices," in *Proc. of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, 2004.

[138] Wang, Y. and Zheng, Y., "Fast and Secure Magnetic WORM Storage Systems," in *SISW '03: Proc. of the Second IEEE International Security in Storage Workshop*, 2003.

[139] Warfield, A., Hand, S., Fraser, K., and Deegan, T., "Facilitating the Development of Soft Devices," in *Proc. of USENIX ATC*, 2005.

[140] Weber, R. O., "Information technology - SCSI Object-Based Storage Device Commands (OSD)." `http://t10.org/ftp/t10/drafts/osd/osd-r10.pdf`, accessed May, 2007.

[141] Weil, S., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C., "Ceph: A Scalable, High-Performance Distributed File System," in *Proc. of OSDI*, 2006.

[142] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C., "CRUSH: Controlled, Scalable, Decentralized placement of Replicated Data," in *Proc. of Supercomputing*, 2006.

[143] Welch, B. and Gibson, G., "Managing Scalability in Object Storage Systems for HPC Linux Clusters," in *Proc. of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004.

[144] West, R., Zhang, Y., Schwan, K., and Poellabauer, C., "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Transactions on Computers*, 2004.

[145] Wetherall, D. J., Guttag, J., and Tennenhouse, D. L., "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Proc. of IEEE OPENARCH*, April 1998.

[146] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T., "The HP AutoRAID Hierarchical Storage System," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, 1996.

[147] Willmann, P., Kim, H., Rixner, S., and Pai, V., "An Efficient Programmable 10 Gigabit Ethernet Network Interface Card," in *Proc. of HPCA*, 2005.

[148] Witchel, E., Larsen, S., Ananian, C. S., and Asanovic, K., "Direct Addressed Caches for Reduced Power Consumption," in *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2001.

[149] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "SmartPointers: Personalized Scientific Data Portals in Your Hand," in *Proc. of Supercomputing*, 2002.

[150] Xu, D. and Jiang, X., "Towards an Integrated Multimedia Service Hosting Overlay," in *MULTIMEDIA '04: Proc. of the 12th annual ACM international conference on Multimedia*, 2004.

[151] Yang, C.-W., Lee, P. C., and Chang, R.-C., "Lyra: A System Framework in Supporting Multimedia Applications," in *Proc. of IEEE International Conference on Multimedia Computing and Systems*, 1999.

[152] youb Kim, H. and Rixner, S., "TCP Offload through Connection Handoff," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[153] Zilles, C. B., Emer, J. S., and Sohi, G. S., "The Use of Multithreading for Exception Handling," in *Proc. of the International Symposium on Microarchitecture (MICRO)*, 1999.

VITA

Himanshu Raj was born in Bareilly, India. He received the Bachelor of Technology degree in Computer Science and Engineering from Indian Institute of Technology, Guwahati, India in 2000, and Masters of Science degree in Computer and Information Sciences and Engineering from University of Florida, Gainesville in fall 2001. After a brief stint with a startup company, he joined the Systems research group at the College of Computing, Georgia Institute of Technology in fall 2002.

Himanshu completed his Ph.D. dissertation entitled "Virtualization Services: Scalable Methods for Virtualizing Multicore Systems" under the guidance of Prof. Karsten Schwan in Fall 2007. His research ranges from Virtualization and Operating Systems support for I/O, memory management, and security in high performance systems, to energy management in mobile and embedded systems.